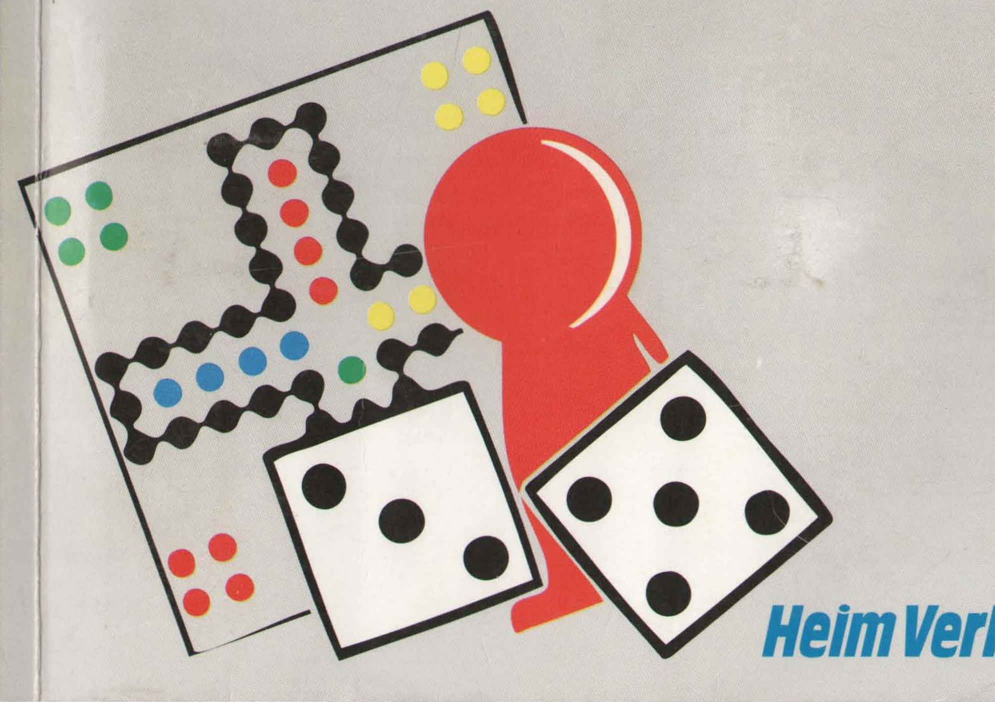
ATARI® ST/STE/TT und FALCON 030

Klaus Dieter Pollack

Spiele selbst programmieren

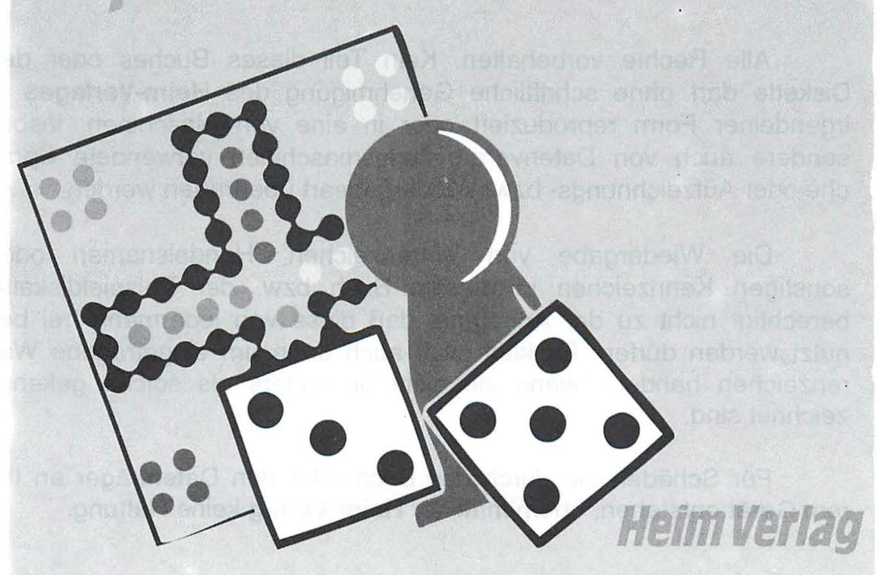


Heim Verlag

ATARI® ST/STE/TT und FALCON 030

Klaus Dieter Pollack

Spiele selbst programmieren



Heim Verlag

Spiele selbst programmieren

Autor: Klaus Dieter Pollack

Darmstadt-Eberstadt : März 1993

Heidelberger Landstraße 194
6100 Darmstadt-Eberstadt
Telefon: 0 61 51 - 94 77 - 0
Fax : 0 61 51 - 94 77 - 18

Druck: Druckerei der **Heim GmbH**, 6100 Darmstadt-Eberstadt

Inhaltsverzeichnis

Vorweg, ein paar Grüße an !

Hans Peter, denk an unsere 8 Bit Zeiten !

Biggi, lerne über Deinen Schatten zu springen !

Jörg M., gib ihr noch eine Chance !

Katja, I O U (seriously)

Thomas, to be in England....

und

Klaus, Matthias, Stefan, Jörg, Daniel und Diana, Axel, Maren Oliver, Arne und Marcus (Amiga Freaks), Gert, Christine, noch 'ne Katja, Sonja, Virginia, Xiaoping, Robert (PC Freak), Regina, u.v.m.

meine Komilitonen vom Studiengang AI in der FH- Frankfurt

"bitte nicht böse sein, wenn ich jemand vergessen habe !"

ATARI, viel Glück für euren Falcon030 !!!
"and may the force be with you"

Vielen Dank an !

Kay, Torsten und Simon

Die Konstrukteure des ATARI 520ST, für das gute Bus-Timing !
Die Konstrukteure der ATARI 400/800 Computer !!!

Die Demoprogrammierer auf dem AMIGA, weil ihre Demos immer neue Herausforderungen brachten.

Vorwort

Vor etwa zehn Jahren hat ein Computer mit der Leistungsfähigkeit eines ATARI ST noch mehr als 20000 Mark gekostet. Als ich Ende 1985 mir meinen ersten ST kaufte, galt der Motorola 68000 noch als sehr schneller Prozessor. Ich war zu dieser Zeit von meiner 8 Bit Maschine, einem ATARI 400/48K in Sachen Geschwindigkeit noch nicht so sehr verwöhnt. Mein ST hat mich zu dieser Zeit sehr oft durch seine Leistungsfähigkeit erstaunt, um allen die so wie ich damals heute umgestiegen sind, einen all zu großen Schock zu ersparen, möchte ich Ihnen jetzt schon mitteilen, daß man Spiele, die z.B. auf dem C64 in Maschinensprache geschrieben worden sind auf dem ATARI ST auch in BASIC programmieren kann ! Die ST Computer sind mittlerweile in Preisregionen gefallen, in denen man auch reine Videospielgeräte findet; doch der ATARI ist mehr. Er ist nicht nur eine gute Videospielkonsole sondern ein komfortables Textverarbeitungssytem, mit dem man auch ohne Festplatte angenehm arbeiten kann, er ist DTP - Center, kann messen, steuern und regeln und ist eine Hilfe für jeden Profimusiker, auch konstruieren, zeichnen und vieles mehr ist mit ihm möglich. Er ist fast viel zu schade nur zum Spielen, außer man realisiert auf ihm die eigenen Ideen und Wünsche. Um Ihnen zu zeigen wie einfach das geht, und um zu zeigen, daß der ATARI ST immer noch ein geniales Gerät ist, auf dem zweitklassige Umsetzungen von Spielen nicht nötig sind, habe ich dieses Buch geschrieben. Dieses Buch richtet sich somit sowohl an den interessierten Neuling wie auf den fortgeschrittenen Programmierer, und nicht zuletzt ist dieses Buch als Nachschlagewerk gedacht, das dem Programmierer von anderen Rechnertypen die Konvertierung ihrer Spiele erleichtern soll.

Teil I: Einführung in die Spieleprogrammierung

Die phantastische Einbildung ist zugleich praktische Zauberin, wird in dem unaufhörlichen Wechselspiel zwischen Phantasie und Idee, zwischen Affektivität und Praxis zur Quelle aller denkbaren Innovationen, welche die menschliche Evolution vorangetrieben und bereichert haben.

Aus, "Das Rätsel des Humanen", von Edgar Morin.

Immer wieder werde ich von irgendwelchen "Kids" besucht, die mit ihrem Computer nichts anderes anzufangen wissen, als einschalten, Spiele laden, ausschalten. Doch gelegentlich ist der Eine oder Andere dabei der mich fragt, "Wie funktioniert die Kiste, und wie kann ich selbst Spiele schreiben ?". Da es meiner Meinung nach (bis jetzt) für die ATARI Rechner noch kein brauchbares Buch zu diesem Thema gibt und ich nicht jedem persönlich zeigen will (und kann) wie man Spiele programmiert, habe ich dieses Buch geschrieben.

Don't Panic !!!

Eine kurze Einführung in die Arbeitsweise des Computers

Heutige Computer bestehen aus **Eingabegeräten** wie:

- Tastatur - Scanner - Joystick - Maus - CD ROM

und aus **Ausgabegeräten** wie

- Bildschirm - Drucker - Plotter u.v.m.

und Geräten die beides können

- Diskettenlaufwerke - MOD Laufwerke - Festplatten ...

bei einigen Computern ist das meiste schon im Gehäuse integriert, bei den Atari Computern meistens nicht.

Was noch fehlt, ist eine Einheit, die diese Ein/Ausgabegeräte steuert. Diese Einheit besteht bei den ATARI Computern aus einer CPU, je nach Gerät ein MC 68000/010/020/030, "etwas" Speicher und einigen Chips, die die Kommunikation mit den entsprechenden Geräten erst möglich machen (sogenannte Peripheriebausteine).
Die CPU (**C**entral **P**rocessing **U**nit, d.h. zentrale Verarbeitungs-Einheit) ist das Herz eines jeden Computers. Sie führt die Befehle, die im Speicher stehen, aus. Wenn man einen ATARI ST einschaltet, springt die CPU automatisch über einen Zeiger, der am Anfang des "Speichers" steht (dazu später mehr), zu einem nichtflüchtigen Speicherbereich und führt dort schon sein erstes Programm aus, bei den meisten Rechnern ist es das sogenannte BIOS (oder Kernel), bis Anfang 1986 galt das auch für den ST, denn seit April 1986 liefert Atari seine Computer mit einem "fast" kompletten Betriebssystem aus. Der nichtflüchtige Speicherbereich ROM (**R**ead-**O**nly-**Memory**, Nur-Lese-Speicher) erhielt seinen Namen, weil er vom Prozessor nur gelesen aber nicht verändert werden kann. Nichtflüchtig ist er, weil er nach dem Ausschalten seinen Inhalt behält. Das ist auch gut so, da sich nach dem Einschalten des

Rechners im RAM (**R**andom **A**ccess **M**emory, Speicher mit wahlfreiem Zugriff) "Müll" befindet, d.h. der Inhalt des Speichers enthält zufällig irgendwelche Daten - der Prozessor würde kein ausführbares Programm finden, was einen Systemcrash (totalen Absturz des Computers) zur Folge hätte. Übrigens, der Zugriff auf das RAM ist "wahlfrei", weil man es sowohl beschreiben als auch lesen kann.

Nun, aber was ist überhaupt Speicher ?

Speicher kann man sich als eine Straße vorstellen. Jedes Haus hat eine Hausnummer, ein Haus entspricht einem Byte, zwei gegenüberliegende Häuser (eins mit gerader Hausnummer und eins mit ungerader) nennt man Wort und zwei Wörter ein Langwort. So ist es zumindest bei den Motorola Prozessoren. Aus dem Informatik-Studium weiß ich, daß das Wort eine Hardware (Prozessor) Eigenschaft ist, d.h. es gibt für andere Prozessoren auch größere oder kleinere Worte. Die Hausnummer nennt man Adresse. Viele Prozessoren können viel mehr Speicher adressieren (verwalten) als die Computer beinhalten, in dem sie stecken. Der 68000er kann z.B. 16,7 Millionen Bytes adressieren. Im Atari werden aber meist nur 1 MB RAM und 192KB ROM davon genutzt. Übrigens, 1 MB bedeutet 1024*1024 Bytes und 1 KB sind 1024 Bytes. Jedes Byte ist in 8 Bit (**BI**nari **D**igi**T**, binäre Einheit) unterteilt; jedes Bit kann man sich als einen Schalter vorstellen, der zwei Zustände (an & aus) annehmen kann. Ist einer dieser Schalter mit einem "Peripheriebaustein" verbunden, kann das Setzen dieses Bits (einschalten) z.B. die Hintergrundfarbe des Bildschirms oder die Bildwiederholfrequenz ändern.

Was macht also ein Programm ?

Ein Programm ist also nichts anderes als eine Menge von gesetzten und gelöschten Bits, die der Prozessor schrittweise (z.B. 16 Bit) abarbeitet. Diese Bits interpretiert der Processor als Befehl, der ihn z.B. das Bit eines Hardwareregisters (Speicherstelle eines Peripheriebausteins) setzen oder löschen läßt. Die CPU kann aber auch einen Teil in einen anderen Teil des Speichers kopieren, Inhalte von Speicherstellen addieren, subtrahieren, multiplizieren und andere schöne Sachen. Näheres dazu finden sie im Kapitel über die Einführung in die Maschinensprache.

Wozu braucht man dann das Betriebssystem ?

("das fragen sich viele gute Demoprogrammierer auch")

Das Betriebssystem ist ein Programm, das sich im ROM befindet. Es ist eine Sammlung von Funktionen, die (meistens) sehr einfach aufzurufen sind. Es sind Funktionen wie z.B. das Laden eines Sektors von Diskette oder das Zeichnen einer Linie zwischen zwei Punkten. Man sollte diese Funktionen (wenn möglich) nutzen, denn nur so kann man sicherstellen, daß ein solches Programm auch auf den ATARI Rechnern der nächsten Generation z.B. dem TT oder Falcon ohne Probleme seinen Dienst verrichtet. Zum Thema Betriebssystem sollte man nicht vergessen, daß die Annehmlichkeiten, wie es das Desktop des Ataris bietet, ein Teil des Betriebssystems sind, welche ab TOS 2.0X auch wirklich als angenehm empfunden werden können (allen anderen empfehle ich Programme wie die GEMINI Shell).

A.1. Kurze Einführung ins GFA - BASIC

Ich habe GFA-BASIC als "Lernsprache" gewählt, weil dieser BASIC Dialekt sowohl flexibel in seiner Anwendung und leicht erlernbar ist als auch strukturiertes Programmieren unterstützt.

Um die Technik der Spieleprogrammierung zu erlernen, möchte ich die grundlegenden Techniken in einer einfach zu erlernenden Progammiersprache vermitteln. Diese Sprache nennt sich BASIC. BASIC ist eine Hochsprache, so wie z.B. C, Modula, Forth, FORTRAN , PASCAL.

Was ist eine Hochsprache ?

Es gibt zwei verschiedene Arten von Hochsprachen, es gibt Interpreter Sprachen und Compiler Sprachen und natürlich Ausnahmen, die beides sind, wie z.B. GFA-BASIC. Eine Compiler Sprache besteht meist aus einer Entwicklungsumgebung mit einem Editor, einem Compiler mit Linker und einer Shell (ein Programm zum Starten des Editors oder Compilers, mit gleichzeitigem übergeben von Dateinamen und anderer Optionen). Ein Editor ist eine abgespeckte Textverarbeitung, in die man den Programmtext eintippt. Dann speichert man den Text auf eine Diskette und ruft den Compiler auf. Der Compiler wandelt den Programmtext in ein ausführbares Programm. Bei einer Interpretersprache hat man in den meisten Fällen einen Editor mit gleichzeitiger Syntaxkontrolle, d.h. nachdem man eine Zeile mit Befehlen eingegeben hat, testet das Programm, ob er die eingebenen Befehle kennt und gibt sofort eine Fehlermeldung aus. Das ist allerdings der Idealfall; viele Interpreter geben die Fehlermeldungen erst nach dem Start des Programms aus. Nachdem man dann das Programm vollendet hat, kann man es sofort starten, vorausgesetzt, es hat keine Strukturfehler.

Nun, was macht die Hochsprache aus den Befehlen ?

Eine Compiler wandelt den Text, den man mit dem Editor erstellt hat, in Maschinensprache um, allerdings ist das nur so eine Art "Ablaufplan" und der Linker verbindet diesen "Ablaufplan" mit den entsprechenden Progammteilen einer Bibliothek. (Das ist nur eine grobe Darstellung, die ich später noch weiter erläutern werde.) Bei der Ausführung des Programms werden die Programmteile der Bibliothek von dem "Ablaufplan" aufgerufen. Beim Interpreter muß das Programm Befehl für Befehl übersetzt und die entsprechenden Funktionen aufgerufen werden. Natürlich kostet das Übersetzen etwas Zeit, auch kann ein Compiler, weil er mehr Zeit zum Übersetzen hat, effektivere Programme erzeugen, so daß eine Compilersprache in der Regel schneller ist als das interpretierte Pendant.

Was ist denn so ein Programm ?

Ein Programm ist ein Algorithmus, d.h. eine Arbeitsanweisung, die in eine Computersprache (und damit vom Compiler oder Interpreter in Maschinensprache) umgewandelt wird.

Was ist ein Algorithmus ?

Ein Beispiel:

Algorithmus zum Öffnen einer Tür

1. Gehe zu Tür
2. Drücke Klinke herunter
3. Ziehe Tür an der Klinke in Richtung Wand

mehr zum Thema Algorithmus finden sie in div. Büchern, die die Einführung in die Informatik behandeln.

Ein einfaches Spiel mit Linien und Kreisen

Es ist extrem einfach in GFA - Basic Linien und Kreise zu zeichen. Die Befehle dazu sind:

Line ax,ay,ex,ey

Wobei die ersten beiden Werte ax/ay den Anfangspunkt angeben, von dem aus die Linie gezeichnet werden soll und ex/ey den Punkt angibt, zu dem die Linie gezeichnet werden soll. Die linke obere Ecke des Bildschirms ist der Punkt (0,0) und die rechte untere Ecke in der niedrigen ST Auflösung (319,199). Diese Werte stimmen, solange sich keine Hardwareerweiterungen wie Overscan, Hyperscreen oder Pixel Wonder in Ihrem Atari befinden. Falls das doch der Fall sein sollte, kann die rechte untere Ecke durchaus höhere X- und Y-Werte aufweisen.

```
Circle ax,ay,r
       I  I I____   Radius
       I  I______   Vertikale Position  (X - Position)
       I_________   Horizontale Position (y - Position)
```

Der Befehl Circle zeichnet einen Kreis.
(ähnl. wie ein Zirkel, den man in der Realität dazu benötigt)

```
Pellipse ax,ay,rx,ry,   <--- wi1,wi2 optional (bei bedarf)
         I I I I         I I_______       Winkel 1 in 1/10°
         I I I I         I_________       Winkel 2 in 1/10°
         I I I I
         I I I I____      senkrechter  Radius
         I I I______ waagerechter Radius
         I I________      Anfangspunkt Y
         I__________      Anfangspunkt X
```

mit dieser Funktion kann man Kreissegmente und Ellipsen erzeugen, wenn man wi1 und wi2 setzt (Anfangs u. Endwinkel).

```
Box ax,ay,ex,ey
        | | | |___     senkrechte Endposition
        | | |_____     waagerechte Endposition
        | |_______     senkrechte Anfangsposition
        |_________     waagerechte Endposition
```

Wenn man jeweils ein 'P' vor die Zeichenfunktionen schreibt wie Pellipse / Pcircle / Pbox, werden die jeweiligen Körper ausgefüllt.

Diese Befehle sind GFA-BASIC spezifisch, rufen jedoch interne Funktionen des Betriebssystems auf. Nähere Informationen über diese Befehle entnehmen Sie Büchern über GFA-BASIC, GEM (Graphics Enviroment Manager - Verwalter der Grafischen Oberfläche) und/oder VDI.

Nun sollten Sie mit den Funktionen etwas herumprobieren. Außerdem die auf der Diskette befindlichen Listings (Programme im Quellcode - also in Worten) LINE.LST / CIRCLE.LST / BOX.LST ausprobieren. Diese werden mit der Funktion MERGE des Interpreters eingeladen. (auch mit der Taste F2 zu erreichen)

Wie kommt das Bild auf den Schirm ?

Der Video-Chip, der Chip, der das Bild erzeugt, tastet "ununterbrochen" einen Teil des Speichers ab, den sogenannten Bildschirmspeicher oder Video-Ram. Dieser wird vom Prozessor oder dem Blitter (einem speziellen Chip der Speicherinhalte kopieren und dabei mit anderen Speicherbereichen verknüpfen kann) beschrieben, je nach Inhalt dieses Speicherbereiches und der Einstellung des Videoprozessors, sieht das was wir auf dem Monitor betrachten können wieder ganz anders aus. Dazu mehr im zweiten Teil des Buches Kapitel B.1

Vorbereitungen für unser erstes Spiel !

Unser erstes Spiel heißt 'Ball gegen die Wand', es ist ein - zugegeben - recht einfaches Spiel, das man aber vor einigen Jahren in ähnlicher Form noch als Spielautomat antreffen konnte.

Steuerung:

Dieses Spiel läßt sich am besten mit der Maus oder einem Paddle steuern. Da ja nicht jeder ATARI ST-User ein Paddle besitzt, habe ich mich für Maussteuerung entschieden.

Dazu stellt uns GFA - BASIC die Befehle

Mousex, *Mousey* und *Mousek* zur Verfügung,

Ein kleines Programm veranschaulicht die Funktion der Befehle.

```
color 1              ! Zeichenfarbe
Repeat               ! Wiederhole
  Plot Mousex,Mousey
Until Mousek=1       ! Bis LinkeMaustaste gedrückt
```

Dieses Programm zeichnet an der Stelle, wo sich der Mauspfeil befindet, einen Punkt. Beim Drücken der linken Maustaste wird das Programm beendet. Das ist sehr wichtig, da man Programme so schreiben sollte, daß man sie regulär verlassen kann, regulär heißt ohne Reset oder Ausschalter. Der Interpreter bietet allerdings auch die Möglichkeit, mit den Tasten [Control] [Shift] [Alternate], alle drei gleichzeitig gedrückt, das Programm zu verlassen.

Die Sache mit den Schleifen !

```
Repeat  <---- Anfang der Schleife
  .
  .
  .
Until Mausek=1  <---- Ende der Schleife
       ( ^ )                |
         |---  Bedingung die nicht !!! erfüllt sein darf,
               damit die Schleife wiederholt d.h. vom
               Anfang bis zum Ende noch einmal !!!
               durchlaufen wird.
```

Was man sich dabei merken muß !

Eine Schleife dieser Art, d.h. das Innere der Schleife, wird mindestens einmal durchlaufen !

Eine solche Schleife nennt man oben offene Schleife, weil bei der Programmausführung die Schleife "auf jeden Fall" von oben betreten und hoffentlich auch irgendwann wieder verlassen wird. Eine Schleife, die nicht verlassen wird (niemals, außer man betätigt den allmächtigen Reset oder verhindert die Stromzufuhr - wie schon erwähnt), nennt man Endlosschleife.

Es gibt jedoch von oben abweichende Schleifen,

hier ein Beispiel:

```
while mousek<>1 <----- Anfang der Schleife
    .        ( ^  )
    .          |
    .          |__ Bedingung welche erfüllt sein muß
    .              damit die Schleife mindestens einmal
    .              durchlaufen wird
    .
    .
wend <----- Ende der Schleife
```

Es kann durchaus passieren, daß eine solche Schleife, während das Programm abgearbeitet wird, niemals durchlaufen wird und nur in ganz speziellen Situationen das Innere einer solchen Schleife überhaupt ausgeführt wird.

Es fehlt jetzt nur noch ein Befehl, um auf Änderungen von bestimmten Werten zu reagieren.

```
repeat
    IF mousek=0
        print "DIE MAUS IST TOT"
    else
        print "DIE MAUSTASTE ist gedrückt"
    endif
until inp(-2)=-1
if mousek=0
^  ^
I  I__ Bedingung erfüllt
I ____Wenn
else <-------- sonst
Endif <------- Ende
```

Die "Entscheidungsanweisung" 'if' 'then' 'else' ist ein grundlegendes Konstrukt vieler Hochsprachen d.h. es ist eine Befehlskombination, die wirklich wichtig ist und so oder so ähnlich auch in Pascal und/oder C (fast überall) Anwendung findet.

Nun zur Erklärung:

Nach 'IF' folgt eine Bedingung (z.B. Mousek=0), falls diese Bedingung wahr ist, (d.h. erfüllt ist - in diesem Fall, keine Maustaste gedrückt worden ist) werden die darunter stehenden Befehle bis zu einem möglichen 'Else' Befehl ausgeführt. Wenn die Bedingung jedoch nicht erfüllt ist (also 'Mousek' den Wert 1,2 oder 3 hat, also eine Maustaste gedrückt ist) dann werden die Befehle ab einem möglichen Else oder einem Endif abgearbeitet.

GFA-BASIC, enthält natürlich noch einige Befehle mehr, doch das Erläutern dieser Befehle würde den Umfang des Buches sprengen und das angestrebte Ziel verfehlen. Wer jedoch Intresse an einer Einführung in GFA-BASIC hat sollte sich ein entsprechendes Buch (siehe Anhang) kaufen. Jedoch müßte man mit der Bedienungsanleitung zu GFA-BASIC und diesem Buch auskommen können.

Außer Befehlen besteht eine Programmiersprache noch aus anderen Dingen.

z.B. Operatoren

Beispiele:

'+'	plus	zur	Addition	(dazu zählen)
'-'	minus	zur	Subtraktion	(abziehen)
'*'	mal	zur	Multiplikation	
'/'	geteilt	zur	Division	

u.v.m

Vergleichsoperatoren (die wichtigsten)

'=' gleich
'<>' ungleich
'>' größer als
'<' kleiner als
'>=' größer gleich
'<=' kleiner gleich

oder Variablen

Variablen sind einigen Lesern schon aus der Mathematik bekannt, es sind Platzhalter für bestimmte Zahlen. Da der Computer jedoch nicht unbegrenzt große Zahlen darstellen kann, gibt es in Basic verschiedene Arten von Zahlen mit denen der Rechner verschieden schnell rechnen kann.

Variablen bestehen aus mindestens einem Buchstaben
z.B. 'a' oder 'b'
oder mehr Buchstaben z.B. 'zahl', 'Wort','tausche'.

Variablen, die nur aus einem Buchstaben bestehen, sind Fließkomma-Variablen (Float - abkürzung für floating Point). Fließkomma-Variablen sind Variablen welche Dezimalbrüche ('Kommazahlen') darstellen können. Sie benötigen 8 Byte Speicher und können Zahlen in einem Bereich von 2.225073858507*10-308 bis 3.595386269725*10+308 darstellen.

Wenn hinter dem Namen der Variblen noch ein Zeichen wie:

! Boolean - kann nur die Werte 0 (False) oder -1 (True) annehmen, benötigt 1 Byte
| Byte - Werte von 0 bis 255 möglich benötigt 1 Byte
& Wort - Werte von -32767 bis +32768 möglich benötigt 2 Byte
% Integer - Werte von etwa +/- 2 Giga
Float - siehe oben
$ Stringvariable - enthält Buchstaben

steht, benötigt der Rechner unterschiedlich viel Speicher und verarbeitet diese unterschiedlich schnell (Faustregel, je weniger Speicher - desto schneller).

Wie benutzt man nun die Variablen ?

Fast wie in der Mathematik,

z.B. a%=a%+1

auch möglich 'INC a%' (INCrementiere d.h. erhöhe)
erhöht den Inhalt von a% um eins

oder

```
n%=5
c%=10
l%=n%+c%*2
```

1. Zeile: n% bekommt den Wert 5
2. Zeile: c% bekommt den Wert 10
3. Zeile: c% wird mit 2 multipliziert und das Ergebnis zu n% addiert und dann in l% geschrieben

l% enthält als Ergebnis die Zahl 25

auch möglich

'ADD v%,10' ;erhöht den Inhalt von v% um 10
'ADD' steht für "ADDiere" also '+'

oder

```
a$="Spiele - "
b$="programmieren"
a$=a$+b$
```

Als Ergebnis enhält a$ die Zeichenkette ('ne Menge Buchstaben)
'Spiele - programmieren'

Auch wenn Sie jetzt nicht alles verstanden haben, wird Ihnen das meiste wohl bei den Kommentaren zu den Listings bewußt werden !

1.1 Das erste Spiel

Ist auf der Diskette zum Buch in 'KAPA1.TOS' unter dem Namen 'FIRST.LST' zu finden.

Das Spiel ist auflösungsunabhängig programmiert (!), d.h. es fragt zuerst im 'work_out Feld' des VDI (Teil des Betriebssystems der für die Bildschirmausgabe zuständig ist) nach, welche maximale horizontale und vertikale Auflösung möglich ist und schreibt diese Werte in die Wortvariablen 'xmax&' u. 'ymax&'

In etwa so !

```
xmax&=WORK_OUT(0)
ymax&=WORK_OUT(1)
```

Danach wird das Unterprogramm 'aufbau' aufgerufen, das eine Box mit maximalen Bildschirmausmaßen zeichnet und die Bildschirmausgabe auf 'XOR' (eXklusiv OdeR) umschaltet. Das bedeutet, wenn man einen Punkt mit einer bestimmten Farbe zeichnet und dann mit derselben Farbe darüberzeichnet, wird der zuerst gezeichnete Punkt wieder gelöscht.

GRAPHMODE 3; schaltet auf XOR

Wahrheitstabelle:

A	B	XOR
0	0	0
0	1	1
1	0	1
1	1	0

Hier die Wahrheitstabelle von 'XOR', die bei der Grafikausgabe auf duo/monochromen Bildschirmauflösungen genau mit gesetzten '1' und gelöschten '0' Bildpunkten übereinstimmt, wobei 'A' dem zu zeichnende Punkt und B dem Bildschirminhalt entspricht (es geht auch umgekehrt) und unter 'XOR' die dann auf dem Bildschirm sichtbaren Punkte wären.

Wichtig ist noch zu wissen, wie das mit der Bewegung des Balls funktioniert !

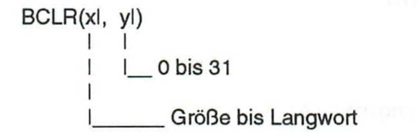
```
' ********* Links und Recht für den Ball *******
  ADD xb&,nx&
  ADD yb&,ny&
  IF nx&=2
    IF xb&>xmax&-10 ! Wenn Ball rechts angekommen
      nx&=-2
    ENDIF
  ELSE
    IF xb&<10 ! Wenn Ball links angekommen
      nx&=2
    ENDIF
  ENDIF
' *************** Fast das gleiche für die Y Richtung ******
  IF ny&=2
    IF yb&>ymax&-10 ! Wenn Ball am unteren Rand angekommen
      init
    ENDIF
  ELSE
    IF yb&<10 ! Wenn Ball am oberen Rand angekommen
      ny&=2
    ENDIF
  ENDIF
```

Eigentlich ganz einfach: Wenn 'xb&' oder 'yb&', die Variablen für den Mittelpunkt des Balls (Kreis), oben bzw. unten angekommen sind, wird die Variable 'nx&' bzw. 'ny&' in positiv bzw. negativ, so daß aus der 'ADD'ition eigentlich eine 'SUB'traktion wird und der Ball seine Richtung ändert, da 'nx&' bzw. 'ny&' die Variablen sind, die die Schrittweite der "Flugbewegung" des Balls angeben. Natürlich springt der Ball, wenn er ganz unten ankommt, nicht nach oben sondern das Programm beginnt von vorn.

Mit dem Betätigen der Taste 'Q' für 'quit' verläßt man das Programm.

UNTIL BCLR(INP(2),5)=81 !verlasse Schleife, wenn Taste = "Q"

wobei das 'BCLR(INP(2),5)' den ASCII - Code der eingegebenen Tasten jeweils (falls es Buchstaben sind) durch Löschen des Bit 5 in Großbuchstaben umwandelt, da 'BCLR' (Bit CLeaR) lösche Bit bedeutet.



bedeutet, daß das Bit yl im Operanden xl gelöscht werden soll.

Man könnte das Programm noch auf vielfältige Weise erweitern, versuchen Sie doch ein kleines Breakout bzw Arkanoid daraus zu machen, in dem sich mit 'BOX' noch einige Vierecke auf den Bildschirm zeichnen lassen und diese, wenn sie getroffen wurden, wegzeichnen.

A.2. 16 Farbige Softwaresprites

Einleitung

Nachdem wir im letzten Kapitel gesehen haben, wie man mit Kreisen und Linien Spiele macht, möchten wir (zumindest ich) natürlich auch mal ein paar hübschere Objekte, wie wir es von professionellen Spielen gewöhnt sind, auf den Rechner bringen.

Diese mehrfarbigen Objekte, die scheinbar unbeeindruckt vom Hintergrund sich hin und her bewegen, nennt man Sprites.

Im Gegensatz zum Amiga, Atari XL, C64, also all jene Computer, die eigentlich vorrangig als 'Spielcomputer' konstruiert wurden, kann die ATARI ST(E)/TT/F030-Serie keine Hardware-Sprites, d.h. Objekte, die wirklich unabhängig vom Bildschirm, auf dem sie sich bewegen, das bedeutet, daß sich das Objekt nicht im Bildschirmspeicher befindet, darstellen.

Das bedeutet allerdings nicht, daß man keine Bilder darstellen könnte, bei denen es den Anschein hat, das Objekt bewege sich ohne Einfluß auf dem Hintergrund.

2.1 Die Theorie dazu !

2.2.1 Was ist 'page flipping' ?

Wenn man es weiß, ist es eigentlich ganz einfach (wie alles bei der Spieleprogrammierung). Der Bildschirmspeicher, den der Videoshifter darstellt, darf sich fast überall im Speicher befinden (beim TT gilt das nur im ST-RAM). Der 'Trick' besteht einfach darin, das Bild in einem Teil des Speichers aufzubauen und währenddessen ein Bild, das sich im anderen Teil des Speichers befindet, darzustellen, d.h. der Videoprocessor zeigt einen Speicherbereich an, während der Prozessor oder Blitter den anderen beschreibt. Würde man kei-

nen anderen Speicherbereich anzeigen, könnte man sich ansehen, wie der Prozessor das Sprite mit dem Bildschirm verknüpft (zusammenrechnet). Die Technik der wechselnden Bildschirme nennt man 'page - flipping' (ein bezeichnender Name !). Man kann es sich wie beim Daumenkino vorstellen, wobei der vom Videoprozessor (Shifter) dargestellte Speicherbereich, den Seiten des bemalten Blocks entspricht. Um Speicher zu sparen, benutzt man bei der Spieleprogrammierung nicht für jede 'Seite des Daumenkinos' ein neues Stück Speicher, sondern benutzt zwei oder drei Bildschirmspeicher, welche man wechselt und in einem der gerade nicht dargestellten Speicherbereiche eines der nächsten Seiten aufbauen läßt.

2.1.2 Der Aufbau der Seiten erfolgt meist in folgender Reihenfolge:

- Zurückschreiben des geretteten Hintergrunds (siehe nächster Punkt)

- Retten des Hintergrundes

  Beim Retten des Hintergrundes wird dieser an den Stellen gerettet (in einen Puffer, ein Teil des freien Speicher, geschrieben), an denen später die Sprites zu sehen sind. Man kann natürlich auch den kompletten Hintergrund in einen Puffer kopieren und dann die Teile des Bildes, die man mit den Sprites überschrieben hat, wieder zurückkopieren.

- Kopieren der Maske auf den Bildschirm

  Die Maske besteht meist nur aus einer Bitplane (aus einem "Bildschirmspeicher" mit nur einer Farbe). Diese wird mit der Funktion 'AND' (eine Art der Verknüpfung) auf den Bildschirmspeicher kopiert (im Kapitel über die Assemblersprache, werde ich die Verknüpfungsarten erklären). Dabei wird jede Bitplane so oft kopiert wie das Zielbild Bitplanes (z.B. 16 Farben 4 Bitplanes -

den 2 hoch 4 ergibt 16) hat. Wenn man Betriebssystemfunktionen zum Kopieren nutzt, kann es helfen, alle Bitplanes auf einmal zu kopieren, was jedoch einen größeren Speicherbedarf für die Maske bedeutet z.B. bei 16 Farben 4 mal so viel Speicher. Da man die Maske aber leicht errechnen kann, fällt das in vielen Fällen nicht ins Gewicht.

Die Maske sieht wie folgt aus, an den Stellen, an den die Bits gesetzt sind, d.h. der Punkt im einfarbigen Raster ist an dieser Stelle zu sehen, ist die Farbe des Sprites 'Hintergrundfarbe'. An der Stelle an der die Bits der Make gelöscht sind, sind auch die Punkte des Sprites ungleich der Hintergrundfarbe. Bei der Verknüpfung der Maske mit dem Bild (unser Hintergrund, auf dem sich der Sprite bewegt) wird dieses an diesen Stellen schwarz d.h. Farbe 0 (alle Bitplanes gelöscht, denn nur dann kann man erfolgreich das Sprite an diese Stelle kopieren).

❒ Kopieren des Sprites in den Bildschirm

Es wird das Sprite, das sich als Bitmap (Bit Mapped - aufgeteilt in ein Bit "Tiefe" Speicherteile) im Speicher befindet - also im selben Format wie das des Bildschirmspeichers - auf die Stelle kopiert, an die wir zuvor die Löcher mit der Maske "eingestanzt" haben. Der Sprite wird dabei nicht einfach so hineinkopiert, weil ein Sprite eben auch ein viereckiges Stück ist (es würde viel zu lange dauern, den Sprite Punkt für Punkt (also Bit für Bit) an die richtige Stelle des Bildes zu kopieren). Um zu vermeiden, daß auch die nichtgesetzten Punkte mitkopiert werden, benutzt man die im Prozessor und Blitter fest "verdrahtete" Funktionen 'OR' oder 'XOR'. Diese beiden Funktionen kopieren an die durch die Maske gelöschten Stellen die Bitplane des Sprites. Das alles geschieht natürlich immer dann, wenn der Bildschirm mit dem man das macht, gerade nicht vom Videoshifter angezeigt wird!

2.2 Die Realisierung !

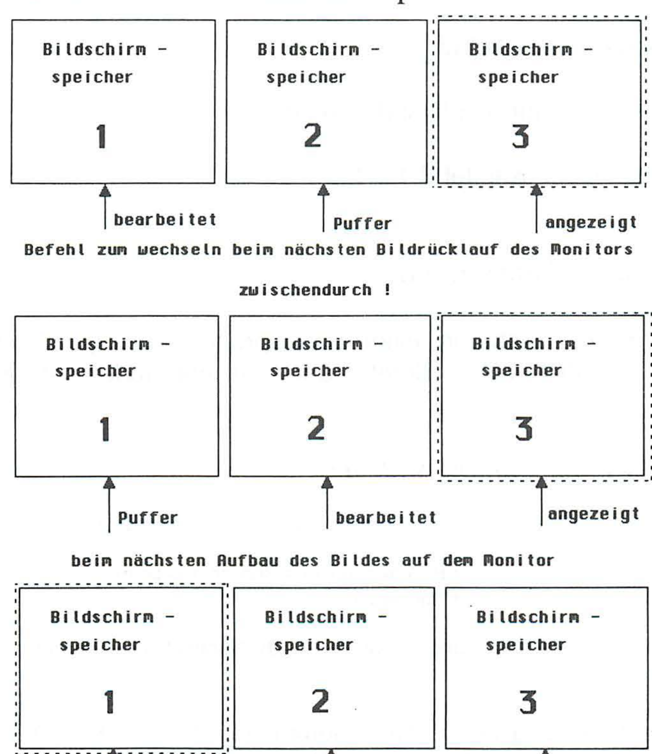
2.2.1 Voraussetzungen

❒ Bildsynchronisation

Der Bildaufbau des Monitors wird zwar vom Computer gesteuert, erfolgt jedoch kontinuierlich (Zeile für Zeile). Der Programmierer hat dabei nur sehr geringe Möglichkeiten (50/60 Hz Umschaltung) Einfluß zu nehmen. Es muß jedoch sichergestellt sein, daß jedes Umschalten der Bildschirmseiten genau dann erfolgt, wenn das Monitorbild neu von oben aufgebaut wird. Wenn man die Umschaltung der Bilder nicht mit dem Monitor synchronisiert, kann es passieren, daß die eine Hälfte des Bildschirms schon das neue Bild darstellt, während auf andere Hälfte das alte Bild zu sehen ist. Um die Synchronisation zu ermöglichen, gibt es viele verschiedene Möglichkeiten. Ich möchte zuerst einmal eine relativ einfache Möglichkeit erklären, die, verglichen zu dem geringen Aufwand, den man bei der Realisierung betreiben muß, schon beachtliche Ergebnisse erzielt. Sie ist vor allem sehr leicht von Hochsprachen zu nutzen und erfordert keine Kenntnisse in Assembler (Programm, das vom Menschen verständliche 'Maschinensprache' in Rechner verständliche umsetzt). Dazu muß man wissen, daß die Register (Videobasisregister - 3 Stück), in denen man die Adresse des darzustellenden Bildspeichers ändern kann, erst dann in den Shifter übernommen werden, wenn das nächste Monitorbild aufgebaut wird. Das kann unter Umständen bis zu 20 ms dauern. Man könnte natürlich, nachdem man das darzustellende Bild errechnet hat, den Bildschirmspeicher umschalten, abwarten bis der Shifter die neue Adresse übernommen hat und dann das nächste Bild vom Rechner aufbauen lassen. Das hat jedoch den Nachteil, daß man sehr viel Zeit damit vergeudet, darauf zu warten, daß der Videoprozessor die nächste Bildschirmseite anzeigt, damit man in die damit freigewordene Seite vom Spieler unbemerkt das nächste Bild aufbauen kann. In Maschinensprache ließe sich der Bildaufbau noch so geschickt programmieren, daß er zeitlich mit dem Anzeigen des nächsten Bildschirms zusammenfallen würde. In einer Hochspra-

che ist das effektiv nicht zu realisieren. Trotzdem muß man nicht unbedingt auf das Umschalten, das während der Dunkelphase des Monitors (Vertikal Blank abgekürzt 'VBL') erfolgt warten. Einfacher ist es, wenn man 3 Bildschirmspeicher benutzt und dann, während der eine Speicher angezeigt wird, den zweiten mit den neuen Bilddaten beschreibt und der dritte darauf wartet, beim nächsten VBL angezeigt zu werden .

Umschalten mit drei Bildschirmspeichern



❒ Das Umschalten der Bildschirmspeicher

Da gibt es 3 Möglichkeiten (aller guten Dinge sind drei)!

(1) Das direkte Beschreiben der Register des Videoschifters (dazu später mehr).

(2) Das Beschreiben der Systemvariable die bei &44e d.h. an der Hexadezimalen (Zahlen zur Basis 16) auch $44e (Assembler) oder 0X044E (in C) geschrieben wird (dazu später mehr).

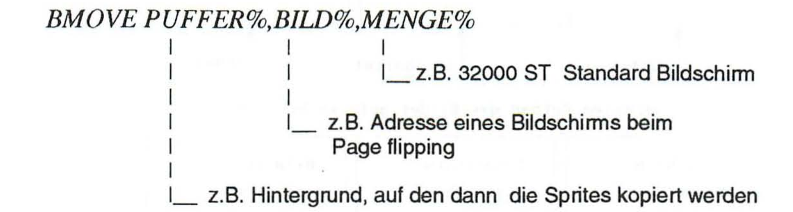
(3) Den Betriebssystem-Aufruf

Void Xbios(5,l:logadr%,l:physadr%,rezl)

Einzelheiten, siehe Kapitel B.1.1.1

❒ Das Retten des Hintergrunds

Da ich die ganze Sache in einer Hochsprache aufzeigen möchte, benutze ich natürlich einen Befehl, der dort vorhanden ist. Es ist der Befehl



'*BMOVE*' funktioniert ähnlich wie 'memmove' in der Programmierspache 'C'. Es kopiert Speicherblöcke, man muß nur Anfangsadresse, Zieladresse und Länge des Speicherblocks angeben, wobei sich der Quellblock und der Zielblock überlappen dürfen.

❐ Das Kopieren der Sprites

Dazu benutze ich wieder eine GFA-BASIC Funktion 'RC_COPY, die ähnlich funktioniert wie '*vro_copyfm*' des VDI.

RC_COPY QUELL%,QUX&,QUY&,BREIT&,HOCH& TO ZIEL%,ZX&,ZY&,VERKNl

---

Quell% = Quelladresse

QUX& = X - Quellposition } linke obere
QUY& = Y - Quellposition } Ecke

Breit& = Breite des Sprites (oder Maske)
Hoch& = Höhe des Sprites (oder Maske)

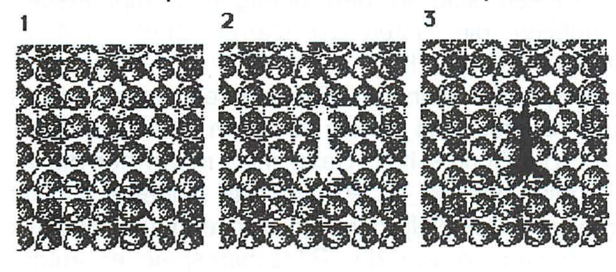
Ziel% = Adresse des Zielbildschirms (Seite beim 'page flipping')

ZX& = X - Zielposition} linke obere Zy& = y - Zeilposition} Ecke

---

VERKNl = 4 für AND (MASKE), 7 für XOR (SPRITE)

Maske und Sprite auf den Bildschirm kopieren




Orginal


Maske

1 Hintergrund zurückschreiben
2 Maske mit dem Hintergrund UND verknüpfen
3 Orginal Raster mit dem Hintergrund ODER verknüpfen

2.3. Sprites ohne Maske

Gelegentlich kann es sinnvoll sein, Sprites ohne eine Maske über den Bildschirm zu bewegen, z.B. an einer Stelle, an der der Bildschirm leer ist oder nur eine Farbe hat oder monoton ist. In diesem Fall kann es sich lohnen, den Hintergrund in den Sprite einzubauen. Diese Technik wird z.B. an bestimmten Stellen bei dem Programm BODO (ich habe mal etwas mit dem Original! eines Freundes herumexperimentiert) benutzt, es ist also durchaus gebräuchlich. Allerdings benutzt man am besten den Replace Modus '1' des *RC_COPY* Befehls. Auch in meinem Beispielprogramm Sprites auf der Diskette werde ich diese Möglichkeit nutzen, wie auch die Animation von Sprites mit einer Maske.

2.4 Animation von Sprites

Bei richtiger Animation bewegen sich nicht nur die Sprites, sondern sie verändern sich auch. Sie laufen, fliegen, springen, fahren u.v.m. Zeichentrickähnliche Grafiken sind in den heutigen Computerspielen durchaus nicht unüblich, die Zeiten der starren Sprites mit fester Form sind wohl endgültig vorbei !!!
Die Realisierung benötigt fast keinerlei Rechnerzeit, man kopiert lediglich die Form des Sprites aus einem anderen Teil des Speichers, in dem man sich die Sprites holt. Dazu muß mann nur die X- und Y-Position der Quelle bei RC_COPY auf ein anderes Objekt richten. Bei der in meinem Beispielprogramm verwendeten Technik ist der Spritespeicher genau so wie der Bildschirmspeicher aufgebaut. Diesen habe ich vorher mit einem Malprogramm gezeichnet, so daß die Sprites sich leicht verändern lassen. Ich habe NEOCROME bzw. NEOCROME MASTER benutzt um die Sequenzen (eine Folge von Sprites) zu zeichen. NEOCROME ist ein Public Domain Programm, also für jeden leicht zu erhalten.
NEOCROME MASTER wird nicht kommerziell vertrieben, es wurde von Demo-Programmierern um einige interessante Funktionen erweitert. Man kann damit z.B. Animationen direkt im Zeichenprogramm testen und es hat eine "tierisch" schnelle Routine zum Drehen von Objekten u.v.m. .

A.3. Der Aufbau des Bildschirmspeichers

oder wie erzeugt die Funktion 'WHITE' aus 'Argon 4' die Maske ?

3.1 Die 3 Bildschirmauflösungen des Atari ST

Hohe (ST) mit 640*400 Punkten Monochrom, d.h. nur eine Bitplane und zwei Farben (Vordergrund und Hintergrundfarbe)

Mittlere (ST) mit 640*200 Punkten 4 Farben d.h. z.B. 2 Bitplanes

Niedrige (ST) mit 320*200 Punkten 16 Farben d.h. vier Bitplanes, die meistbenutzte Auflösung für Spiele

Der TT hat zudem noch drei weitere Auflösungen

Hohe (TT) 1280*960 wie (ST HOCH)

Mittlere (TT) 640*480 Punkte wie ST niedrig

Niedrige (TT) mit 320*480 Punkten 256 Farben d.h. acht Bitplanes wohl die zukünftige Auflösung für Spiele auf dem TT

Der Falcon 030 hat einige neue Auflösungen und unterstützt die TT-Auflösungen nicht komplett.

Mögliche Auflösungen des Falcon 030:

Normal	Overscan	Farben
640*400	768*480	2/4/16/256
		65535 (nur Interlace)
320*400	348*480	4/16/256/65536
320*200	320*240	4/16/256/65536
640*200	768*240	2/4/16/256/65536

zusätzlich sind alle ST Auflösungen möglich .

3.1.1 Die Hohe Auflösung (ST)

In der 640*400 Auflösung ist der ATARI ST sehr augenschonend. Mit einer Bildwiederholfreqenz von 71Hz kommt er den von verschieden Organisationen geforderten 100Hz Monitoren viel näher als andere Computer in seiner Preisklasse. In dieser Auflösung kann der Atari ST (leider) nur noch zwei Farben (schwarz und weiß) darstellen. Der Bilschirmspeicher besteht aus nur einer Bitplane, aber was bedeutet das ? Das bedeutet, daß die ersten 16 Punkte des Bildschirms auch im ersten Wort (2 Byte bzw. 16 Bit) Platz finden. Ist ein Bit an, das bedeutet auf 1 gesetzt, dann hat dieser Punkt nicht mehr die Bildschirmfarbe (Hintergrund), sondern, da es nur zwei Farben gibt, die Farbe von gesetzen Punkten. Natürlich stehen in Wirklichkeit keine Einsen und Nullen im Speicher, es sind nur geöffnete und geschlossene Schalter und wie jedes Gerät reagiert der Computer auf diese Schalter. In unserem Fall reagiert der Videoshifter auf offene Schalter mit der Hintergrundfarbe und auf geschlossene mit einem gesetzten Punkt.

3.1.2 Die Mittlere Auflösung (ST)

In dieser Auflösung kann ein 'normaler' Atari bis zu 640*200 Punkten mit 4 Farben darstellen, man schreibt im allgemeinen 640*200*2 wobei man Zahl '2' am Schluß Bittiefe nennt, diese Art der Schreibweise benutzt man z.B. bei speziellen Grafikmaschienen. Mit zwei Bitplanes kann man bis zu vier Farben (drei und die Hintergrundfarbe) darstellen. Der Grund dafür ist Relativ einfach, da man zwei Bitplanes nur auf vier unterschiedliche Arten zusammenstellen kann !

```
      zweite Plane _______  _______ erste Plane
                          I  I
Farbe 0 (Hintergrund)     0  0
Farbe 1                   0  1
Farbe 2                   1  0
Farbe 3                   1  1
```

Nun sind die Planes nicht hintereinander in einem Byte oder einem Wort, sondern:

1.Wort	1.Plane	für 16 Pixel
2.Wort	2.Plane	für 16 Pixel

Diese Auflösung wird nur für einige wenige Spiele genutzt, z.B. schalten einige Adventures (z.B. The Pawn) an der Stelle, an der die Grafik, die 16 oder mehr Farben enthält, endet, die Auflösung um, damit sie für den Text die 80 Zeichen pro Zeile, die in der mittleren Aulösung leicht möglich sind,nutzen können.

3.1.3 Die Niedrige Aulfösung (ST)

Diese Auflösung wird wohl am meisten für irgendwelche Spiele genutzt, da man relativ viele Farben (16) zur Verfügung hat, und mit einigen Tricks auch alle 512 bzw. 4096 Farben gleichzeitig darstellen kann (dazu später mehr). Die Auflösung 320*200*4 übertrifft bei weitem das, was ich mir zu meinen 8 Bit Zeiten vorstellen konnte, ist jedoch heute zu Zeiten von VGA und Super VGA nichts Besonderes mehr. Es gab jedoch 1985 (als der ST veröffentlicht wurde) weder VGA noch Super VGA Karten. Der Standart war Hercules-Monochrom- und CGA-Grafikkarten, die kaum mit den Auflösungen des ST mithalten können. Doch die Zeiten ändern sich und so die Standards. Die Speicheraufteilung der niedrigen ST-Auflösung ist im Prinzip die der mittleren, außer daß zwei Bitplanes mehr zur Verfügung stehen.

Farben	Bitplanes
Farbe 0 (Hintergrund)	0000
Farbe 1	0001
....	....
....	....
Farbe 15	1111

Reihenfolge der Planes im Speicher !

1.Wort	1.Plane für 16 Pixel
2.Wort	2.Plane für 16 Pixel
3.Wort	3.Plane für 16 Pixel
4.Wort	4.Plane für 16 Pixel

3.1.4 Die Hohe Auflösung (TT)

Diese ist vergleichbar mit der hohen ST-Auflösung, allerdings mit einer Auflösung von 1280*960 Punkten, die wiederum mit einer Bildwiederholfrequenz >70 Hz auf den entsprechenden Monitor gebracht werden.

3.1.5 Die Mittlere Auflösung (TT)

Ist vom Aufbau mit der niedrigen ST Auflösung vergleichbar, allerdings mit 640*480 Punkten, was der Standard VGA-Auflösung entspricht.

3.1.6 Die Niedrige Auflösung (TT)

Eine meiner Lieblingsauflösungen auf dem TT, da bis zu 256 Farben (ohne Tricks) gleichzeitig möglich sind. Mit Tricks sind sogar (fast ohne jegliche Einschränkungen) 4096 Farben gleichzeitig möglich. Die 320*480*8 Auflösung übertrifft sogar die beim PC für Spiele übliche 320*200*8 (MCGA) Auflösung. Sie ist für mich die! Auflösung für Spiele auf dem TT und ich hoffe, daß sie bei den nächsten Umsetzungen von PC auf Atari nicht unberücksichtigt bleiben wird.

Farben	Bitplanes
Farbe 0 (Hintergrund)	00000
Farbe 1	00001
....	.....
Farbe 255	11111

Reihenfolge der Planes im Speicher !

1.Wort 1.Plane für 16 Pixel
2.Wort 2.Plane für 16 Pixel
....
8.Wort 8.Plane für 16 Pixel

3.1.7 Die Auflösungen des Falcon 030

Die Bitplanes sind beim Falcon, bis auf eine Ausnahme, genau so angeordnet wie bei den ST/STE/TT Auflösungen. Die Ausnahme ist die "True Color" Auflösung des Falcon !

In den "True Color"-Auflösungen wird keine Palette benötigt, da die Farbe aus den 16 Bit des Bildpunktes bestimmt wird.

Ein Wort im Bildschirmspeicher:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	R	R	R	R	R	G	G	G	G	G	X	B	B	B	B	B

R = rot

G = grün

B = blau

X = niederwertiges (zählt am wenigsten) grünes Bit, das auch für Genlock verwendet wird.

wenn X gelöscht	Transparent (externes Video Signal)
wenn X gesetzt	Overlay (Falcon Video Signal)

Zu dem True Color Modie kommen wir noch im Kapitel B.1 im zweiten Teil des Buches.

3.2. Wie erstelle ich daraus die Maske ?

3.2.1 Sinn einer Maske

Wie SIe im vorherigen Kapitel erfahren haben, ist es Sinn und Zweck einer Maske, ein Loch für die Sprites zu erzeugen, um mit den möglichen Arten der Verknüpfung, welche Hardwareeigenschaften (Prozessor, Blitter) sind, das Bitmuster des Sprites so mit dem Hintergrund zu verknüpfen, das weder Hintergrund noch Sprites verfälscht werden.

3.2.2 Voraussetzung

Die Maske muß an der Stelle, an der irgendein Pixel gesetzt ist auch gesetzt sein, damit auch nur die Stellen des Zielbitfelds (Hintergrunds) gelöscht werden, an denen auch wirklich die Pixel des Sprites gesetzt sein sollen. Die Maske ist im Regelfall so beschaffen, daß nur die Pixel, die im Sprite gesetzt sind, auch in der Maske gesetzt sind.

3.2.3 Erzeugung

Um die Maske zu erzeugen, nimmt man die 'OR' Funktion des Prozessors oder des Blitters und "verodert" alle Bitplanes. Bei der 'OR' Funktion werden alle Bits gesetzt, die im Quell -Zielbitfeld gesetzt sind. Das bedeutet, wenn auch nur irgendein Bit gesetzt ist, also irgendein Punkt irgendeine Farbe hat, daß das entsprechende Bit auch in der Maske gesetzt wird.

3.2.4 Die Verwendung

Ich könnte jetzt die Maske entsprechend der Menge Bitplanes, die ich habe, mehrmals mit dem Bildschirm verknüpfen. Diese Technik würde sich mit Betriebssystemroutinen nur schwer mit der nötigen Performance realisieren lassen. Um dieses Problem zu lösen, schreibe ich die Maske schon von zu Beginn in alle Bitplanes, was zwar viermal so viel Speicher benötigt, aber wesentlich einfacher zu handhaben ist. Bei eigenen Spriteroutinen mit eigenen Bitblock-kopierroutinen sollte man jedoch diese Möglichkeit Speicher zu Sparen nutzen, ergo die Maske einzeln in alle Bitplanes kopieren !!!

A.4. Ein komplettes Spiel

Ich gebe zu, daß das Spiel nicht mehr ganz dem 16 Bit Standard entspricht und ich mir keine besonderen Gedanken über den Spielwitz gemacht habe, es ist ja auch keineswegs meine Absicht gewesen, ein aufregendes interessantes Spiel zu programmieren.

Mir war es eigentlich wichtig ein Spiel zu programmieren, das möglichst anschaulich einige Verfahrensweisen im Bezug auf Spieleprogrammierung aufzeigt. Das Spiel ist auf allen Rechnern schnell "genug", obwohl es zu 98% in GFA-BASIC geschrieben ist, die Maschienenroutinen wurde nur zum Erzeugen der Maske, des digitalen Sounds und natürlich zur Umschaltung der Farben zwischen Score und Spielfeld, verwendet. Nun, es ist eben nur ein Beispiel, natürlich ließe sich mit etwas mehr Maschinensprache und einem guten Konzept ein Ballerspiel programmieren, das etwas länger Spaß macht. Aber mir war es wichtig, so wenig Assemblerroutinen wie möglich zu benutzen, um zu zeigen, daß man nicht unbedingt auf die kompliziertere Assembler-Ebene heruntersteigen muß, um brauchbare Animation und relativ flüssiges Scrolling zu realisieren, denn mit ein paar guten Ideen, etwas Kreativität und Grundwissen über den Aufbau der Hardware kann man manchmal mehr erreichen als mit dem optimiertesten Stück Assemblercode. Es gibt auch noch Probleme beim Starten von 'Argon 4' mit 'MULTI TOS', da ich zum einen nur eine Beta Test Version von 'MULTI TOS' zur Verfügung hatte (trotzdem vielen Dank an Atari Deutschland) und zum anderen ein schnelles Action-Spiel in einer Hochsprache allein schon eine Herausorderung ist, wenn man sich dann noch an die Konventionen eines Multi-Tasking Systems halten muß (von dem man noch nicht genau weiß, wie es später, wenn es nun mal endlich richtig fertig ist, mal aussehen wird) es also im Prinzip unmöglich ist. Trotzdem habe ich schon einige Vorkehrungen getroffen, die eine höchst mögliche Kompatibilität versprechen, aber dazu später mehr.

4.1 Wie fange ich an ?

Am Anfang ist die Idee, die schreibt man sich am besten gleich auf. Wenn mehr als nur eine Person an einem Programm arbeitet, unterteilt man das Programm in mehrere sinnvolle Abschnitte, z.B. einer programmiert die Routine, die die Aliens bewegen, der anderer die Figur des Spielers und das Scrolling der Landschaft. Gut ist auch eine Unterteilung nach Können, denn nicht jeder kann alles gleich gut programmieren. Ein Beispiel hierfür ist, bei 'Argon 4' die PSG Soundroutine hat Kay Römer geschrieben, denn er hat Erfahrung mit Soundchip und Soundroutinen und programmiert gut optimierten Assemblercode! Also habe ich ihn so lange "angebettelt", bis er sich dazu bereit erklärt hat, für mich die Routine für den "YAMA" Soundchip zu schreiben. Auf diesem Wege "Danke Kay !".
Aber nicht nur Programmierer werden beim Spieldesign benötigt, auch Zeichner und Musiker. Da ich selbst schon lange Zeit Musik mache und gelegentlich auch zeichne, habe ich das bei 'Argon 4' selbst übernommen (verzeiht mir).
Ok, nehmen wir an, die Idee, die Musik, die Sounds und die Grafik sind ausgearbeitet. Ihr wißt daß alles realisierbar ist. Dann würde ein "Vollblut Informatiker" die benötigten Module (Procedueren) auf dem Reißbrett entwerfen. Das bedeutet, man schreibt sich erst einmal auf, welche Module man alle braucht.

Beispiel:

Joystick_Abfrage
Fragt den Joystick Port A ab
Wenn oben, wird X erhöht
Wenn unten, wird X erniedrigt
u.s.w

Spielersprite
Bewegt je nach Wert von X und Y
An den Rändern wird das Sprite geklippt(abgeschnitten)

Gegersprite
Bewegt sich von rechts oben nach links unten
Verschwindet, indem es sich auflöst.

Wichtig ist, gerade bei mehreren Bearbeitern eines Programms, die Übergabeparameter, d.h. die Daten, die das eine Unterprogramm dem anderen übergibt, in ihrer Form (z.B. Byte, Wort, Integer, String, Bool), festzulegen.

Zusätzlich beschreibt man die Funktion der Module in einer Pseudo Computerspache, einer sogenannten PDL. Bei der PDL (**P**rogramm **D**esign **L**anguage) wird die Funktion des Moduls erklärt, wobei auch Strukturelemente und Schleifen benutzt werden, aber die restlichen Befehle umgangssprachlich hingeschrieben werden.

Beispiel:

Modul: *Sprite_bewegen*

IF *Sprite außerhalb der Ränder*

klippe Sprite (abschneiden an den Rändern, so daß es nicht weitergezeichnet wird).

zeichne geklipptes Sprite

ELSE

zeichne ganzes Sprite

END

Wenn man alle Module so definiert hat, erstellt man eine Modulhierachie, d.h. man "malt" Kästchen, die miteinander mit Pfeilen verbunden werden, die aufzeigen, wer wen aufruft und welche Module untergeordnet sind.

"Richtig" gute Informatiker (solche Leute, die Computer und ihre Programmierung studiert haben) betreiben sogar noch ein bißchen mehr Aufwand, aber belassen wir es hierbei.

Nun, wer dann soweit ist, ist einem fertigen Spiel schon sehr nahe, aber wer als Anfänger kommt schon soweit, das artet ja in Arbeit aus. Das erste Spiel sollte leicht zu programmieren sein und man sollte in jeder Stufe der Schöpfung einen Fortschritt sehen können. Das läßt sich nur dann realisieren, wenn man das Programm wie eine Zwiebel aufbaut, Schale für Schale. Zum Beispiel könnte man bei einem Ballerspiel, zuerst den Fighter zeichnen und ihn dann über den Bildschirm bewegen. Dann den Hintergrund entwerfen und dann den Fighter über den Hintergrund bewegen, den man seinerseits sich wieder bewegen lassen könnte. Dieses Prinzip hat einen höllischen Nachteil, wenn man das Programm schlecht dokumentiert (also nicht immer dahinter schreibt, was man macht, sollte man immer machen) steigt man irgendwann nicht mehr durch (!) und man muß alle Module so offen programmieren, daß man im Notfall noch etwas dazu packen kann, man weiß ja nie was an dieser oder jener Stelle noch dazugefügt werden muß. Aber, wie schon erwähnt, hat man bei dieser Art zu programmieren immer wieder ein Erfolgserlebnis, was ganz schön aufputschen kann. Bei kleineren Programmen ist die "Zwiebelschalen" Methode die motivierendere Art zu programmieren ! 'Argon 4' liegt an der obere Grenze eines Programmes, das mit dieser Art zu programmieren realisierbar ist. Bei größeren Projekten verzettelt man sich sonst allzusehr. Gerade wenn mehrere Personen an einem Projekt programmieren, sollte ein Konzept und die Schnittstelle zwischen den Programmteilen feststehen, z.B.

ein Programmteil spielt die Musik, dann könnte man festlegen,

a) Wie übergeben wird welches Musik Stück gespielt wird ?
z.B. als Zeiger (eine Variable die als Inhalt die Startadresse eines bestimmten Speicherbereichs enthält) auf die Noten enthält oder es wird ein String mit Noten u.v.m. übergeben.

b) In welchem Format die Noten vorliegen ?
z.B. im XBIOS(32) also in einem Betriebssystem eigenen Format.

c) Wieviel Rechenzeit steht dem Rest des Programms noch zur Verfügung während die Musik spielt ?
z.B. nur noch 10% bei mehrstimmigem, digitalem Sound (dann geht sonst fast nichts mehr) oder noch 95% bei Soundchip Musik.

d) Wieviel Speicher darf die Abspielroutine in Beschlag nehmen ?
z.B. eine Routine, die dann den Rest des Programms nachlädt, darf fast den gesamten freien Speicher (sofern nötig) für sich in Anspruch nehmen und ihn nach dem Abspielen eines solch langen Samples frei geben und der Haupt-Applikation zur Verfügung zu stellen.
Allerdings, wenn die Abspielroutine nur ein Teil des Programmes ist und vielleicht noch wärend des Spiels läuft, muß sie dem Rest des Programms genügend Speicher übrig lassen bzw. muß sichergestellt sein, daß für den Sound noch genügend Speicher frei ist.

Auch sollte man sich mit dem "Grafiker" einig sein wieviel Farben er nutzen darf und welche, nicht das ein Programm mit 16 Farben laufen soll und der Typ malt mit 256 oder er malt tatsächlich mit 16 Farben, hat aber kein einziges Farbregister für irgenwelche Hintergünde übriggelassen. Worüber man sich vorher einig ist kann es später keinen Streit geben (!) am besten alles aufschreiben und jedem ein Exemplar der Übereinkünfte ausdrucken lassen.

4.2. Das Programmieren des Spiels

Man sollte die Programmteile, wie schon oben erwähnt, so offen wie möglich gestalten, da man sie dann auch in anderen Programmen benutzen kann. Beispielsweise läßt sich die Soundroutine von 'Argon 4' auch in einem von Deinen Programmen verwenden. Wenn man Programme in Programmteile zerlegt, sammelt sich mit der Zeit eine ganze Bibliothek von Modulen an, die selbst wenn das Projekt, für die das eine oder andere Modul geschrieben wurde, ein Fehlschlag war, bei der Realisierung des nächsten Spiels von Vorteil sein kann.

4.2.1. Die Animation in Argon 4

Stelle aus Argon 4 in Procedure '*MAIN*'

```
' ********** setzt neue Bildschirmadresse (pageflipping)
VOID XBIOS(5,L:scmem%(flipl),L:scmem%(flipil),-1)
BMOVE scrollup%+s_off%,scmem%(flipl),28960
```

Genau diese Stelle ist das Herz der Animation !

In der indizierten Variable 'scmem%()' sind die Zeiger auf die drei Bildschirmspeicher, die ich zum Pageflipping (siehe auch die letzten beiden Kapitel) benutze, gespeichert. Das Umschalten geschieht an anderer Stelle in der Procedure 'MAIN' und zwar vor der eigentlichen Umschaltung (Listing oben). Der Videoshifter setzt die neue Adresse erst kurz vor dem Bildaufbau des nächsten Monitorbildes, also volkommen synchron. Der Aufruf 'XBIOS(5)' beschreibt den Videochip mit der als nächstes zu setzenden Adresse und versorgt das Betriebssystem mit der neuen Adresse des (mit VDI, Line A etc.) logischen Bildschirmspeicher, auf den es schreiben soll. Da diese beiden Adressen nicht übereinstimmen, sieht man erst bei jeder dritten (!) Umschaltung den zuletzt beschriebenen Bildschirmspeicher.

Aus Procedure '*MAIN*'

```
INC flipl ! Zähler für das "Page Flipping"
 IF flipl=4
  flipl=1
ENDIF
```

(Weit) Hinter der Umschaltung findet das Setzen von 'flipil' statt.

Von Procedure '*MAIN*'

```
flipil=flipl
```

Durch diese Technik ist Flackern (fast) ausgeschlossen.

Aus Procedure '*MAIN*'

```
RC_COPY maske%,1+offset&,1+kapl,21,longl TO  scmem%(flipl),x&,y&,1
RC_COPY sprite%,1+offset&,1+kapl,21,longl TO  scmem%(flipl),x&,y&,7
```

Wie man sieht, findet die Variable 'flipl' auch bei den Raster-Kopierbefehlen Anwendung, in dem oben gezeigten Listing wird der "Fighter" von dem Bild, das sich an Adresse 'sprite%' befindet auf den gerade aktuellen nicht angezeigten (!) logischen Bildschirmspeicher kopiert. Zuvor wird allerdings seine Maske auf dem Bildschirmspeicher kopiert, um die Stelle, an der das Sprite hin soll, frei von Bildpunkten des Hintergrunds zu bekommen, da es sonst bei den Verknüpfungen der Bitplanes zu anderen Farben kommt als die, die man beim Zeichnen verwendet hat. Die Variable 'kapl' gibt an welche Phase des Fighters zu sehen ist (fliegt er nach links oder rechts), 'offset&' gibt an, ob ein heiler oder ein kaputter Fighter gezeichnet werden soll und 'longl' ob er mit oder ohne Düsenflackern gezeichnet werden soll.

Mit BMOVE scrollup%+s_off%,scmem%(flipl),28960

in der Procedure *'MAIN'* wird der zu scrollende Hintergrund in den logischen Bildspeicher kopiert. Das geschieht natürlich bevor die Sprites hingezeichnet werden. Sonst würden diese ja sofort überschrieben werden. Im Prinzip funkioniert das wie beim Malen mit Öl - Farben, das zuletzt Gezeichnete ist immer das Oberste, nur antrocknen lassen muß man die Farbe nicht.

4.2.2 Das Scrollen des Hintergrundes

Das Scrollen des Hintergrundes ist beim vertikalen Scrolling eine relativ einfache Sache! Da ich sowieso dauernd den Hintergrund neu zeichnen muß, da das in Basic schneller geht als alle von den Sprites überdeckten Stellen neu zu zeichenen, kann ich natürlich auch jedesmal einen anderen bzw. verschobenen Hintergrund übertragen. Nun habe ich mir, bevor ich die Procedure 'MAIN' starte, eine größere Menge Speicher für das Scrolling reserviert und dann aus geladenen Bildern diesen Speicher mit entsprechenden viereckigen Stücken (wie bei einem Puzzle) gefüllt. Da ich die Grundlage für die Bilder zum Erstellen des Scrollspeichers so frei wie möglich definiert habe, kann ich mit einem 32k großen Bild einige 100k Scrollspeicher füllen. Ich verschiebe also die Adresse, von der ich den Hintergrund in einen als Bildspeicher vorgesehenen Bereich, immer um 160 Bytes (nur im 1.Level, im 2. Level 320 Bytes usw.), was einer Zeile in niedriger Auflösung entspricht. Obwohl das im Prinzip Feinscrolling ist, ruckelt die Sache ein klein wenig. Das kommt durch den unterschiedlichen Rechenaufwand des Rechners je nach Spielverlauf. Wenn zum Beispiel viele Aliens auf dem Bildschirm erscheienen, dauert es länger, bis der Rechner auch den nächsten der drei Bildschirme schalten kann und somit dauert es auch länger, bis ich die nächsten 160 Bytes dazuaddieren kann... und wieder umschalten kann. Programme, die vollkommen in Assembler geschrieben wurden, können das Scrolling mit dem VBL synchronisieren, z.B. könnte man den Offset zum Beginn des Scrollspeichers alle zwei VBL's um 160 Bytes erhöhen und schon hätte man relativ gleichmäßiges Scrolling.

4.2.3 Animation der Objekte

4.2.3.1 Die Bewegungen der Aliens

In Procedure *alien1* und *alien2*

Zeichnen der Aliens in Procedure *q_alien1* und *q_alien2*

Das Bewegen der Aliens funktioniert prinzipiell genauso wie das Bewegen des Balls von unserem ersten Spiel im Kapitel 1. Nur das in 'STAGE 1' die Aliens, wenn am unteren Rand angekommen, nicht nach oben abprallen sondern aus dem unteren Rand herausfliegen, um am oberen Rand wieder aufzutauchen. Wenn ein Alien von 'STAGE 1' aus dem unteren Rand verläßt, muß es geklippt werden, d.h. Teile, die über den Rand des Spielfelds überstehen, dürfen nicht gezeichnet werden, da sie sonst an der Stelle erscheinen, wo die Punktzahl (Score) angezeigt wird. Da an dieser Stelle der Hintergrund nicht andauernd redrawed (neu gezeichnet) wird, würde ein Alien an dieser Stelle die hübsche Anzeige (mit satten 14 neuen Farben) am unteren Bildrand teilweise mit einem Alien überdeckt werden; das sieht dann nicht gerade professionell aus.
Die Aliens von 'STAGE 2' bewegen sich genauso wie der Ball vom Spiel von Kapitel 1. Bei jedem abgeschossenen Alien wird ein Boolean (Bitvariable) auf False gesetzt. Per Zufallsgenerator wird "entschieden" ob wieder ein neues Alien erscheinen soll. Bei jedem neuen Erscheinen eines Aliens wird eine Variable hochgezählt, denn nach einer im Programm festgelegten Anzahl von Aliens erreicht man den nächsten STAGE.

Wichtige Variablen:

- Index

all

Die Bytevariable 'all' gibt den Index für die meisten der in Aliens benutzten Variablen. Da alle Aliens einer Stage denselben Algorithmus zur Bewegung benutzen, wäre es wenig sinvoll gewesen für

jeden Alien die Bewegungsroutine erneut aufzuschreiben. So habe ich die wichtigsten Variablen der Bewegung indiziert und lasse mehrmals den gleichen Programmteil durchlaufen, es wird bei jedem Durchlauf der Schleife mit der boolschen Variablen *'init!(al\)'* getestet ob das Alien existiert, also ob es nicht schon abgeschossen oder noch gar nicht erschienen ist, und gegebenenfalls gezeichnet.

- Animation des Alien

ani&(all)

- Position des Alien

X Position alx&(all)
Y Position aly&(all)

- existiert das Alien ?

init!(all)

4.2.3.2 Die Bewegung des Mutterschiffs

In Procedure *mother*

X Position mx&
Y Position my&

Das Mutterschiff bewegt sich immer zum Fighter hin, wobei das Mutterschiff sich ständig nach links und rechts bewegt. Wenn der Fighter sich horizontal bewegt, folgt ihm das Mutterschiff. Das Mutterschiff folgt langsam dem Fighter, je nachdem ob er sich oberhalb oder unterhalb des Mutterschiffs befindet und versucht, ihn zu zerquetschen.

4.2.4 Kollision der Objekte

4.2.4.1 Kollision von Schuß mit Alien oder Mutterschiff

Diese Kollision wird in den Proceduren *mother,alien1,alien2* getestet. Die Kollision wird durch Vergleichen von Rechtecken, die den Ausmaßen der Objekte entsprechen, getestet. Überschneiden sich die Rechtecke zweier Körper, kommt es z.B. zu einer Explosion. Da ein Schuß eine Breite von Null hat, brauche ich in diesem Fall nur die Berührung einer Geraden mit einer Fläche zu prüfen, was mindestens ein Vergleich weniger im *'IF THEN'* zur Folge hat. Es werden jeweils alle drei Schüsse auf Berührung mit einem Alien bzw. dem Mutterschiff getestet, daher die *'FOR - NEXT'* Schleife.

Kollision von Alien mit Schuß:

colx&, coly&, caly& werden zuvor berechnet um beim 'IF THEN' Zeit zu sparen.

```
     .
     .
colx&=alx&(all)+10
coly&=aly&(all)+10
caly&=aly&(all)-10
     .
     .
' ***** Kollision mit Schuß
FOR nl=1 TO 3
  IF shoot!(nl)=TRUE
    IF shx&(nl)=>alx&(all) AND shx&(nl)<=colx& AND shy&(nl)>=caly& AND shy&(nl)<=coly&
      shoot!(nl)=FALSE
      alcol
    ENDIF
  ENDIF
NEXT nl
```

Mutterschiff mit Schuß:

colx&, coly&, wedelx& werden zuvor berechnet um beim 'IF THEN' Zeit zu sparen.

```
    .
    .
coly&=my&+38
colx&=mx&+wedell+130
wedelx&=mx&+wedell
    .
    .
FOR nl=1 TO 3
  IF shoot!(nl)=TRUE
    ' treffer ?
    IF shy&(nl)<coly& AND shx&(nl)<colx& AND shx&(nl)>wedelx& AND y&>coly&
      IF shx&(nl)>wedelx&+49 AND shx&(nl)<wedelx&+60
        ' An der schwachen Stelle ?
        ' ******* Treffer in Mother Sound
        saml=4
        rplay_sound
        ' ******************************
        SLPOKE &H44E,gegner% ! ändere logischen Bilschirmspeicher
        sberx&=shx&(nl)+263-wedelx&-50
        WHILE POINT(sberx&,140-ymothl)=0 AND mothexpl!=FALSE ! Farbig ?
          ' wenn nicht farbig
          IF ymothl>13 ! Sehr tief ins Mutterschiff eingedrungen ?
            mothexpl!=TRUE
            motl=132
            animatl=1
            ' ******* Mother, Explosionsgeräusch
            saml=1
            rplay_sound
            ' ***************
            ymothl=0
          ELSE
            INC ymothl
          ENDIF
        WEND
        ' ********* Striche in schwacher Stelle
        COLOR 0
        LINE sberx&,140-ymothl,sberx&,138-ymothl
        COLOR 1
        SLPOKE &H44E,maske2%
        LINE sberx&,140-ymothl,sberx&,138-ymothl
        shoot!(nl)=FALSE
        ymothl=0
      ELSE
        shoot!(nl)=FALSE
```

```
        ' ****** Sound, Mother nicht getroffen
        saml=2
        rplay_sound
        ' ************
      ENDIF
    ENDIF
  ENDIF
NEXT nl
```

4.2.4.2 Kollision des Fighters mit Alien und Mutterschiff

Diese Kollision wird in den Proceduren mother,alien1,alien2 getestet.

Kollision mit Mutterschiff:

Auch hier wird zuvor (siehe 4.2.4.1) *colx&, coly&* berechnet !

```
IF x&+11>mx& AND x&+11<colx& AND y&+19>my& AND y&<coly&
  colission
ENDIF
```

Kollision mit Alien:

```
IF x&+11>alx&(all) AND x&+11<colx& AND y&+19>aly&(all) AND y&<coly&
  colission
  alcol
ENDIF
```

4.2.4.3 Kollision des Fighters mit dem Hintergrund

Diese Kollision wird in der Prozedure *main* getestet

Im Gegensatz zu der zuvor benutzten Methode, wobei ich die Kollision durch Vergleich der Ausmaße der Körper getestet habe, teste ich hier auf eine bestimmtes Farbregister. Falls also der Fighter an einem der drei Punkte des Fighters die ich auf Berührung mit einer Farbe prüfe, das Farbregister Nummer neun berührt, wird die Procedure Kollision aufgerufen, die erst dann eine Kollision zuläßt,

wenn die Animation beendet ist, die anzeigt, daß der Fighter getroffen wurde, indem er die Boolche Variable 'hit!' auf 'true', was 'wahr' bedeutet, setzt. Eine Kollision zu prüfen wäre in diesem Fall auf eine andere Art auch schlecht möglich, da das Programm ja nie genau weiß, wo sich die gefährlichen Objekte im Hintergrund befinden.

```
IF POINT(x&+10,y&)=9 OR POINT(x&+20+small&,y&+15)=9 =>
    => OR POINT(x&,y&+15-small&)=9 AND hit!=FALSE
  colission
ENDIF
```

In der Procedure *'colission'* wird dann ein Flag gesetzt, also eine Boolsche Variable, die angibt, ob ein Ereignis eingetreten ist und mit dem PSG ein "brummiger Rüttelsound" gespielt.

4.3 Joystick und Keyboard

4.3.1 Die Abfrage des Joysticks

Es wird immer nur der Joystick Port Nr.1 abgefragt.

JOYSTICK POSITION:

```
5  1  9
 \ | /
4 - 0 - 8
 / | \
6  2  10
```

Was man bei der hier gezeigten schematischen Darstellung der ausgegebenen Werte beim Drücken in eine bestimmte Richtung beachten muß, ist das System bei der Anordnung der Zahlen. So ist links=>4 + oben=>1 links&oben=>5 , so daß man beim genauen Hinschauen bemerkt, daß die Seiten links, rechts, oben, unten jeweils einem Bit zugeordnet sind,

Bit 0, oben
Bit 1, unten
Bit 2, links
Bit 3, rechts

und die schrägen Joystickpositionen jeweils eine 'OR' Verknüpfung der Richtungsbits der geraden bedeutet. Wenn man mal einen Joystick öffnet und z.B. die Betätiger (z.B. Mikroschalter) von oben und unten gleichzeitig betätigt erhält man eine 3, was wiederum bedeutet, daß effektiv 16 (4 Bit) Joystick Positionen statt 9 möglich wären, wenn man einen entsprechenden Joystick bauen würde. Man könnte allerdings die zusätzlichen Verknüpfungen der Tasten auch mit Funktionen wie Pause oder ähnl. belegen.

In Procedure '*main*'

```
SELECT STICK(1)
 CASE 0
  in_mitte
 CASE 1
  oben
  in_mitte
 CASE 2
  unten
  in_mitte
 CASE 4
  links
 CASE 8
  rechts
 CASE 5
  links
  oben
 CASE 9
  rechts
  oben
 CASE 6
  links
  unten
 CASE 10
  rechts
 unten
ENDSELECT
```

In den Proceduren 'links', 'rechts', 'oben', 'unten', werden die entsprechenden Variablen zur Steuerung des Fighters verändert.

4.3.2 Tastenabfrage

Ich hatte eigentlich geplant, die Abfrage der Tasten über das BIOS "laufen zu lassen" , mit der GFA - Basic Funktion 'INP(2)', die BIOS(2), also 'Bconin', benutzt. Beim Testen mit MULTI -TOS mußte ich feststellen, daß ich diese Funktion nicht so schnell abfragen kann, daß das AES die eingegebene Taste nicht erhält. Da aber das AES weiterhin im Hintergrund läuft und gelegentlich ein Tastendruck zur Ereignisverwaltung durchkommt, weil ich den Puffer nicht schnell genug leeren kann, bekomme ich natürlich diese

Taste nicht gemeldet, da das AES mit dem Auslesen der Taste diese aus dem Tastaturpuffer löscht, was in diesem Fall bedeuten würde, daß die Tastaturabfrage unter MULTI - TOS mal funktioniert und mal nicht funktioniert. Daher habe ich entschieden, die komplette Tastaturabfrage über das AES abzuwickeln, was zwar durch die langen Funktionsaufrufe etwas mehr Rechenzeit in Anspruch nimmt, als wenn man das mit dem BIOS regelt, aber auf jeden Fall auch unter MULTI - TOS funktioniert.

Wobei die Abfrage der Tasten etwa so erfolgen muß !

zuerst

```
ON MENU KEY GOSUB xyz
```

wobei dieses Richten der AES Funktion auf eine Unterroutine, in diesem Fall 'xyz', vor dem Eintritt in eine Schleife zu erfolgen hat, die dann

```
REPEAT
 ON MENU
UNTIL ...
```

das ON MENU sorgt für das ständige Abfragen, ob ein AES Event aufgetreten ist. Wenn dieser EVENT aufgetreten ist, verzweigt das Programm bei dem *ON MENU zu 'xyz'*

dort kann man in *MENU(14)* in den Bits 0..7 den ASCII Code finden und in den Bits 8..15 den Scancode.

4.4 Verwaltung von Highscore Liste

Das Verwalten einer High Score Liste ist im Prinzip eine recht einfache Sache. Man hat´also eine Liste von geordneten (!) Werten und einen Wert, den man in die Liste einordnen will. Man testet jeden Wert, der in der Liste steht, mit dem Wert, den man einordnen möchte. Wenn man einen kleineren Wert in der Liste findet als den

den man einordnen möchte, wird der Rest der Liste um eine Stelle in der Liste nach unten geschoben und der Wert, mit dem man verglichen hat, an die Stelle des kleineren Werts geschrieben. Natürlich dürfen bei High Score Listen die Namen nicht fehlen, diese müssen natürlich dann genau so wie die entsprechenden Scores in der Liste verschoben werden.

```
> PROCEDURE high_score !Testet ob Score in High Score
  WAVE 0,0
  ch_palette
  p_cls
  ' ****** Welcher Platz ? *********
  FOR tl=1 TO 10
    EXIT IF high%(tl)<score%(mitspl)
  NEXT tl
  ' ****** Innerhalb ? **********
  IF tl<11
    ' ******* Rest um einen Plaz nach unten *****
    FOR nl=10 DOWNTO tl
      high%(nl+1)=high%(nl)
      high$(nl+1)=high$(nl)
    NEXT nl
    ' *********************************
    high%(tl)=score%(mitspl) !score in Liste eintragen
    high$(tl)=SPACE$(20) !neuen Eintrag löschen
    h_o_fame
    taste!=FALSE
    posl=0
    DEFTEXT tl+3
    ON MENU KEY GOSUB high_name
    REPEAT
      ' ******** Blinken des "Cursors" ********
      IF blink!=TRUE
        blink!=FALSE
        TEXT 50+posl*8,50+tl*8," "
      ELSE
        blink!=TRUE
        TEXT 50,50+tl*8,high$(tl)+" "
      ENDIF
      ' ****************************************
      ON MENU
    UNTIL taste!=TRUE
  ELSE
    h_o_fame
    w_key
  ENDIF
RETURN
```

4.5 Das Abspielen der Musik

Nähere Einführung zu den hier beschriebenen Effekte im Kapitel B.4 zu Sounderzeugung !

Wie man unschwer bemerkt, spielt das Spiel, nach dem alle Programmteile eingeladen wurden, eine hübsche kleine Musik. Wie man bei mir leicht vermutet, habe ich dazu eine eigene Abspielroutine in BASIC geschrieben, die es aber in ihrer Leistungsfähigkeit mit vielen Maschinenroutinen aufnehmen kann. Diese ist so geschrieben, daß man sie gegebenenfalls ohne größere Probleme auch in Maschinensprache umsetzen könnte. Natürlich hat das Schreiben einer solchen Abspielroutine in BASIC einen Haken und der ist zum einen die massig verschlungene Rechenzeit und zum anderen, daß die Routine schon fast in Höchstgeschwindigkeit läuft, noch schnelleres Abspielen würde die Qualität der Effekte wie Vibrato etc. unerträglich (zumindest für mein zartes Gehör) machen. Wem die Musik, die meine kleine Routine abspielt, nicht gefällt, kann via 'COMPOSE.LST' in dem File 'KAPA4.TOS', eine eigene Musik komponieren. Dieser kleiner Composer ist natürlich kein anwenderfreundliches, mausgesteuertes Programm (noch nicht), sondern ein kleines Tool, mit dem man durch Eingabe von Datazeilen Musik komponieren kann, die dann z.B. von der Abspielroutine in ARGON 4, die fast identisch mit der in 'COMPOSE.LST' ist, abgespielt werden kann.

4.5.1 Die Funktionen der Abspielroutine

Die Abspielroutine ist dazu fähig, dreistimmigen PSG Sound abzuspielen und unterstützt auch das Rauschen in unterschiedlicher Tonhöhe.

Frequenz: (16 Bit Wert)

Stimmung, mittleres C = 440Hz

	1. Oktave	2. Oktave	3. Okt.	4. Okt.	5. Okt.	6. Okt.
C	3395	1697	849	424	212	106 (105)
C#	3201	1600	800	400	200	100
D	3020	1510	755	377	189	94
D#	2850	1425	712	356	178	89 (90)
E	2690	1345	672	336	168	84 (83)
F	2544	1272	636	318	159	79 (80)
F#	2406	1203	601	300	150	75
G	2267	1133	566	283	141	70
G#	2140	1070	535	267	134	67 (66)
A	2020	1010	505	252	126	63
A#	1904	952	476	238	119	59 (60)
H	1800	900	450	225	112	56

Die Werte in der 1. Oktave und die in der 6. Oktave in Klammer sind mit den Ohren und einem Synthesizer getestete Werte. Alle anderen Werte berechnen sich wie folgt:

n te Oktave/2 = n-1 te Oktave

Da ich alle Werte schon ausgerechnet habe, muß man diese nur ablesen. Diese Werte erzeugen auch als 12 Bit-Periodendauer, bei Direktprogrammierung des PSG, die jeweiligen Noten. Ist die Frequenz=>9000 wird die (Frequenz-9000)*256 als Rauschfrequenz interpretiert, falls das Bit für Rauschen gesetzt ist.

Stärke: (8 Bit)

Gibt die Stärke der Veränderung, die bei jedem Aufruf durch 'EVERY n GOSUB xyz' erfolgt, des Tones durch die gesetzten Effekte an.

Effekte: (8 Bit Wert)

Bit 0, Frequenz erhöhen

Bit 1, Frequenz niedriger

Bit 2, Lautstärke größer (nicht bei Ringmodulation)

Bit 3, Lautstärke kleiner (nicht bei Ringmodulation)

Bit 4, Vibrato (Wellenform Dreieck)

Bit 5, Ringmodulation

Bit 6, Reserviert

Bit 7, Rauschen (nicht bei Ringmodulation)

Werden mehrere Bits gleichzeitig gesetzt, tragen diese Effekte gemeinsam zur Klangbeeinflussung bei.

Lautstärke (Velocity bzw. Amplitude): (8 Bit)

Es sind Werte von 0-15 sinnvoll, gibt normalerweise die Lautstärke des gespielten Tones einer Stimme an. Wenn eine Stimme ringmoduliert wird, gibt die Lautstärke die Art der Modulation an (Ausnahme Lautstärke = 0), wobei

0 = kein Ton (alle Stimmen aus)
1 = Hüllkurve = 12, Geschwindigkeit der Hüllkurve = Frequenz
2 = Hüllkurve = 14, Geschwindigkeit der Hüllkurve = Frequenz
3 = Hüllkurve = 14, Geschwindigkeit der Hüllkurve = Frequenz/48

Wie sich die Ringmodulation anhört, kann man nicht beschreiben, das muß man einfach ausprobieren !

4.5.2 Das Format meiner Song (.sng) Files

Byte 0 - 7, Magic 'ST-MODUL'
Byte 7 - 15, Modul - Name
Byte 16, Menge der Tracks bis 255

Danach "Noten" im Format

Stimme 1: Frequenz, Stärke, Effekt, Lautstärke
Stimme 2: Frequenz, Stärke, Effekt, Lautstärke
Stimme 3: Frequenz, Stärke, Effekt, Lautstärke

pro Track 32 mal

4.5.3 Die Programmierung der Soundroutine

So eine Soundroutine ist eine relativ komplizierte Sache, ich habe selbst etwa eine Woche gebraucht ("ich habe zwischendurch auch mal was anderes getan als Programmieren"), um die für Argon 4 fertigzustellen. Insofern möchte ich nicht jedes Detail dieser Routine, die ja nur zur Anschauung dienen soll, erklären, sondern grundsätzliche Techniken erklären.

4.5.3.1 Abspielen der Musik parallel zum Programm

Ein Programm ist normalerweise eine sequentiell abgearbeitete Reihe von Befehlen.

"Wie soll denn da was parallel laufen ?"

Da wir im Regelfall nur eine CPU in unserem ATARI haben und nur einen Bus zum Speicher, auf den nur ein Prozessor "gleichzeitig" zugreifen kann (ausgenommen FALCON030), ist es nicht möglich, eine Abfolge von Befehlen parallel zum Programm laufen zu lassen.

Es gibt jedoch die Möglichkeit, den Prozessor kurzzeitig zu unterbrechen und ein paar weitere Befehle ausführen zu lassen, die nicht zum Hauptprogramm gehören, und dann am Hauptprogramm weiterarbeiten zu lassen. Dies geschieht beim ATARI schon bei jedem Tastendruck oder der Bewegung der Maus, da in diesem Fall der Tastaturprozessor (IKBD) via MFP (verwaltet die Interrupts und mehr) einen Interrupt auslöst und eine Routine ein Datenpaket vom IKBD abholt und verarbeitet, so daß z.B. der Mauspfeil beim nächten VBL (der auch einen Interrupt d.h. eine Unterbrechung auslöst) an neuer Position auf den Bildschirm gezeichnet wird. Dieses kurzzeitige Unterbrechen des Prozessors kann man auch selbst hervorrufen, in dem man den MFP, der auch Timer (wie ein Wecker) Baustein ist, dazu gebraucht, den Prozessor alle paar Millisekunden zu wecken (daran zu erinnern), etwas anderes zu tun. In unserem Fall macht dies das GFA - BASIC, das wiederum, wenn ein solcher Interrupt auftritt, ein Flag setzt und nach jedem BASIC Befehl testet, ob der Interrupt schon aufgetreten ist, also das Flag schon gesetzt ist, um dann in die angegebene Prozedure zu verzweigen.

Die Syntax zum Setzen eines solchen Interrupts in GFA - BASIC sieht so aus,

```
EVERY n% GOSUB xyz
```

(wird mittels den 200Hz Sytem - Timer über den etv_timer $400 Vektor realisiert)

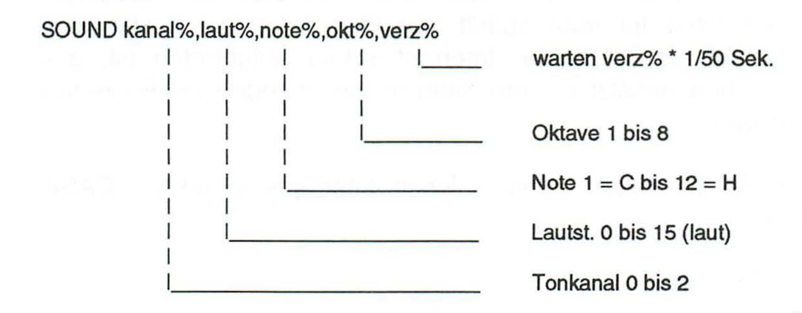
wobei die Zeit durch 'n%' eingestellt werden kann, wenn 'n%=1' dann wird theoretisch alle 1/200stel Sekunden ein Aufruf der Prozedure 'xyz' vorgenommen. Die Zahl, die für 'n%' eingesetzt wird, sollte nicht kleiner als vier sein. Aber selbst bei Werten größer vier kann es unter Umständen, wenn die aufgerufene Procedure "extrem" lang ist, zu Problemen kommen, da es möglich ist, daß die von 'EVERY ...' aufgerufene Procedure selbst wieder unterbrochen wird und es durch Überschreiben von globalen Variablen oder einem Stacküberlauf zu unvorhersehbaren Ergebnissen kommen kann. Ich habe diese Problem für meine Soundroutine, die doch

recht lang ist, gelöst, indem ich immer eine kleine Prozedure aufrufe, die ein Flag setzt, das ich wieder durch ständiges Abfragen teste und entsprechend in meiner Soundroutine aufrufe.

Damit man das ständige Aufrufen einer Procedure auch stoppen kann, bietet GFA - BASIC den '*EVERY STOP*' Befehl an.

4.5.3.2 Das Abspielen mit dem Soundchip

Es gibt in GFA - BASIC zwei Befehle, um den Soundchip zu programmieren, diese basieren beide auf dem Dosound-Aufruf (XBIOS 32) des Betriebssystems. Die zwei GFA - BASIC Befehle nennen sich '*SOUND*' und '*WAVE*'



hinter der Lautstärke kann anstatt Oktave und Note die Frequenz angegeben werden (siehe Tabelle in 4.5.3.1) in der Form '*#*', gefolgt von einer Variablen oder Konstanten als Frequenz bzw. Vorteiler für den Soundkanal. Näheres dazu im Kapitel über Sounderzeugung.

WAVE stim%,huell%,form%,dauer%,verz%

stim%	Kanal an/aus	Bit 0, Kanal 1 an Bit 1, Kanal 2 an Bit 2, Kanal 3 an
	Rauschen an/aus	Bit 3, für Kanal 1 Bit 4, für Kanal 2 Bit 5, für Kanal 3

dazu kann das 256 mal 0..31 addiert werden, um das Rauschfrequenzspektrum festzulegen, was soviel bedeutet wie wenn im oberen Byte des Wortes Werte von 0 bis 31 stehen, diese die Rauschfrequenz angeben.

huell% gibt an, welcher Kanal von der Hüllkurve beeinflußt wird. Belegung wie die untersten 3 Bits von 'stim%'.

form% gibt die Hüllkurvenform an, siehe Kapitel 4.2.1

dauer% 16 Bit-Wert in 0.128 ms Schritten, gibt die Länge der Hüllkurve an.

Beispiel:

```
SOUND 0,15,#1000    !Kanal 1 setzen
SOUND 0,15,#1001    !Kanal 2 setzen
SOUND 0,15,#1002    !Kanal 3 setzen
WAVE 7,7,13,10000   ! Alle an, Hüllkurve für alle, /---, 1,28 Sek.
```

Spielt einen hübschen, langsam lauter werdenden dreistimmigen Klang; durch die sehr nah aneinander liegenden Frequenzen hört man eine starke Phasenverschiebung. Wichtig, immer erst 'SOUND' Befehle für die zu spielenden Kanäle setzen und dann erst mit dem 'WAVE' Befehl abspielen lassen.

4.6 Lesen der Cookies

Ich stelle zuerst die Adresse der Cookies fest, falls diese Null ist, lösche ich ein Flag mit Namen 'cookies!'. Falls die Adresse aber ungleich Null ist, wird das Flag gesetzt, und die Cookies werden langwortweise in einen String verfrachtet und über eine *'SELECT .. CASE'* Struktur auf für das Programm "wichtige" Inhalte geprüft und die Werte in Variablen mit Struktur Namen gespeichert (siehe dazu auch das Kapitel zur Kompatibilität).

```
cookies%=LPEEK(&H5A0) !stellt Adresse der cookies fest

> PROCEDURE cookies !Auswerten der cookies
  IF cookies%=0
    cookies!=FALSE
  ELSE
    cookies!=TRUE
    t%=cookies%
    WHILE LPEEK(t%)<>0
      text$=""
      text$=MKL$(LPEEK(t%))
      tt%=t%+4
      SELECT text$
      CASE "_CPU" ! Welcher Prozessor
        cpu%=LPEEK(tt%)
      CASE "_FPU" ! Floating Point ?
        fpu%=DPEEK(tt%)
      CASE "_MCH" ! welche Maschine ST-TT-Falcon ?
        mch%=DPEEK(tt%)
      CASE "_SND" ! welcher Soundchip ?
        snd%=LPEEK(tt%)
      CASE "_VDO" ! welche Videohardware
        vdo%=DPEEK(tt%)
      ENDSELECT
      ADD t%,8
    WEND
  ENDIF
RETURN
```

Ich teste z.B. über die Cookies, welcher Soundchip sich in dem Gerät, auf dem das Programm gerade läuft, befindet, um dann die DMA -Soundroutine zu laden und benutzen, anstatt der entsprechenden Routine für den PSG .

4.7 Vorbereitungen für MULTI - TOS

Ein Problem bei der Programmierung von ARGON 4 waren die ständigen Änderungen, die ich einerseits wegen MULTI - TOS und anderseits wegen dem Falcon030 vornehmen mußte. Betreffen die Änderungen wegen des Falcon nur das Sound - Subsystem, das sich wider Erwarten ohne vorherige Initialisierungen keineswegs, wie in den Medien angedeutet, kompatibel zum STE/TT DMA Sound verhält. Das MULTI TOS hingegen ist wider Erwarten kein EVENT-orientiertes Multi-Tasking (das Ablaufen mehrerer unterschiedlicher Programme/Routinen im schnellen Wechsel) sondern ein Einteilen der Prozessorzeit in Zeitscheiben, so daß via Interrupt von einem zum anderen Prozess umschalten kann, so daß man das MULTI TOS nicht so einfach daran hindern kann, das Umschalten der Prozesse sein zu lassen. Da ich nicht geplant hatte, ein Fenster zu öffnen, und ich trotzdem nicht wollte, daß ein anderes Programm in den Bildschirmspeicher schreibt, habe ich diesen dauerhaft mit *FORM_DIAL(1)* für andere AES Prozesse, mit *FORM_DIAL(3)* die Drop Down Menüs gesperrt und die Maus für meine eigenen Zwecke reserviert. Zuvor habe ich mit *FORM_DIAL(0,...)*, den kompletten Bildschirm für meine Zwecke reserviert.

```
xres&=WORK_OUT(0) !Maximale X Position
yres&=WORK_OUT(1) !Maximale Y Position
VOID FORM_DIAL(0,0,0,xres&,yres&,0,0,xres&,yres&)
VOID WIND_UPDATE(1)
VOID WIND_UPDATE(3)
```

Wenn ich das Programm beende, teile ich dem AES mit *WIND_UPDATE* mit, daß mein Bildaufbau abgeschlossen ist und gebe mit *WIND_UPDATE(2)* die Mauskontrolle wieder ans AES oder einer anderen laufenden Applikation zurück. Mit *FOR_DIAL(3,...)* gebe ich den reservierten Bildschirmbereich wieder frei und das AES sendet an alle laufenden Applikationen, daß sie die Fensterinhalte wieder auffrischen sollen.

```
VOID WIND_UPDATE(0)
VOID WIND_UPDATE(2)
VOID FORM_DIAL(3,0,0,xres&,yres&,0,0,xres&,yres&)
```

Mit diesen Maßnahmen verhindere ich, daß die Drop Down Menüs herunterklappen oder eine andere Applikation in den Vordergrund kommt und in den Bildschirmspeicher schreibt. Alle grafischen Prozesse sind gesperrt, weil ich die Ausgabe auf den Bildschirm für andere Programme gesperrt habe, diese somit auf das Ende meines Programms warten müssen.

4.8 Anpassung des Programms an die Geschwindigkeit des Rechners

Es gibt mehrere Möglichkeiten, ein Programm an die Geschwindigkeit des Rechners anzupassen; ich habe bei ARGON 4 eine relativ einfache gewählt !

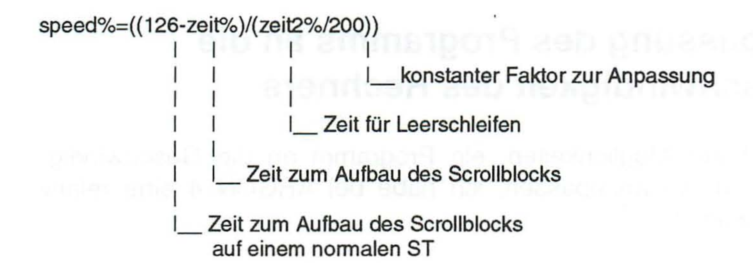
Die Procedure 'scroll' erzeugt im Speicher einen etwa 320 Pixel breiten aber mehrere Bildschirme langen Block, dieser wird dann mit dem 'BMOVE' Befehl jeweils als Hintergrund in einen der drei Pufferspeicher, von denen einer zu sehen ist, kopiert. Zum Erzeugen dieses BLocks wird der *RC_COPY* aus GFA - Basic benutzt. Da ich diesen Befehl auch zum Kopieren der Sprites benutze, messe ich die Geschwindigkeit dieses Vorgangs und die Geschwindigkeit, die der Rechner für 20000 Leerschleifen benötigt (in Procedure 'speed'). Das Messen der Zeit von Leerschleifen und Blockaufbau erfolgt über den Befehl '*TIMER*' welcher die Systemzeit seit dem Einschalten in 1/200 Sekunden zurückgibt und wahrscheinlich *BIOS(6)* also 'Tickcal' benutzt.

Handhabung:

```
zeit%=TIMER
 .
 .        ! Das wird gemessen
 .
 .
zeit%=TIMER-zeit%
```

'zeit%' enthält dann die benötigte Zeit in 1/200 Sekunden.

In der Procedure 'main' wird eine Leerschleife ausgeführt, die die Menge der ausgeführten Durchläufe aus dem Quotienten der gemessenen Zeiten bei '*scroll*' und '*speed*' errechnet wurde.



Die Formel habe ich so aufgestellt, daß bei einem normalen ST ohne Blitter '*speed%*' einen Wert kleiner gleich Null enthält. Diese Art der Geschwindigkeitsanpassung hat unter einem Multitasking Betriebssystem einen großen Nachteil, da die nicht benötigte Rechenzeit durch die Leerschleife "verheizt" wird und nicht anständig genutzt wird. Daher sollte man idealer Weise *EVNT_TIMER* also *AES 24* benutzen und eine Wartezeit in Millisekunden angeben, diese Funktion teilt die nicht benötigte Rechenzeit z.B. einem Accessory zu. Da mein Spiel vom AES des MULI - TOS eher gestört wird, da ja alles so gleichmäßig wie möglich ablaufen soll und mein Spiel keine "reguläre" GEM-Applikation ist, lasse ich die Leerschleife im Programm.

Eine andere Möglichkeit, Spiele rechnergeschwindigkeitsunabhängig zu schreiben, ist, die Bilschirmausgabe mit dem VBL zu syncronisieren, d.h. erst nach dem nächsten VBL (oder mehreren fest definierten VBL's) die Sprites und den Hintergrund neu zu plazieren. Diese Methode ist nur bei extrem maschinennaher Programmierung sinvoll und für mich (bei ARGON 4) nicht in Frage gekommen. Diese Methode hat jedoch das Problem, daß die Bildwiederholfrequenz schon bei normalen ST's unterschiedlich sein kann (50Hz, 60Hz und 71Hz), also das Spiel schon dann schneller oder langsamer wird, was mir bei einigen Spielen auch schon aufgefallen ist. Extrem auffällig ist dieses Problem beim Abspielen von Musik im VBL, wenn man das entsprechende Programm zum Abspielen mal bei 70Hz startet. Hier ist die Lösung, den 200Hz Timer oder andere freie Timer zur Syncronisierung des Programms, nicht nur beim Abspielen von Musik, zu verwenden.

4.9 Die Maschinenroutinen

Ich brauche für dieses Spiel, wie schon erwähnt, vier Maschinenroutinen. Die Routinen sind alle Postition Independent d.h. sie können an jede beliebige Stelle im Speicher geladen und gestartet werden, mit Ausnahme der DMA Routine, diese läuft nur im ST - RAM. Ich werde an dieser Stelle die Maschinenroutinen nicht komplett erklären, da ich dem Leser erst im nächsten Kapitel die nötigen Kenntnisse in Maschinensprache vermittele. Im zweiten Teil des Buches werde ich mich näher mit der direkten Programmierung der Hardware beschäftigen, mit dem Inhalt des zweiten Teils dürfte die Funktion der Routinen dann "hoffentlich" klar werden. Naja, die sind auch nicht ganz so einfach zu verstehen, trotzdem habe ich nichts dagegen wenn jemand sie in eigenen Programmen benutzt, selbst wenn er nicht versteht, wie sie funktionieren. Die Routinen sind natürlich alle als dokumentierte Listings auf Diskette.

4.9.1 Erzeugen der Maske

Die einfachste Routine war eigentlich mal für ein Zeichenprogramm, das ich schreiben wollte, gedacht.

Was macht sie ?
Diese Routine holt sich mit *Logbase (XBIOS 3)* die logische Bildadresse, das ist die Adresse, auf die alle LINE A-Aufrufe erfolgen. In einer Schleife werden dann die vier Bitplanes des Bildschirms 'or' verknüpft, invertiert und auf den Bildschirm zurückgeschrieben, Wort für Wort. Diese Routine ist nicht besonders optimiert, aber dafür leicht verständlich.

Wie benutzt man sie:
- man kopiert die Sprites, für die man die Maske braucht, in einen Speicher
- setzt mit 'setscreen' 'Logbase' auf diesen Speicher
- und ruft die Routine mit einem 'call anfangsadresse%' auf.

```
BMOVE sprite%,maske%,32000
VOID XBIOS(5,L:maske%,L:-1,-1)
CALL white% ! Assembler Teil zum Erzeugen der Maske
```

4.9.2 Umschalten der Palette

Wie man sieht, existieren bei ARGON 4 zwei Paletten, die Umschaltung erfolgt während des VBL und an der 180. Zeile. Dabei werden 14 Register (alle außer Hintergrund und Farbe 0) gewechselt.

Was macht sie ?

Installiert eigene VBL Routine, die dann die ursprüngliche VBL-Routine aufruft und TIMER B setzt, um an Zeile 170 einen Interrupt auszulösen. Initialisiert TIMER B via *Xbtimer (XBIOS 31).*
Die TIMER B Interruptroutine wartet auf Zeile 180 und setzt die Palette. In einem Programm, das diese Routine nutzt, muß das Interrupt Vektor Register auf Automatic End of Interrupt gestellt sein.

```
vr%=&HFFFFFA17   !Interrupt Vektor Register
SPOKE vr%,BCLR(PEEK(vr%),3) !Automatic End of Interrupt
```

Wie benutzt man sie ?

- man dimensioniert eine Variable für RCALL, z.B.: DIM reg%(16)
- man rette alte VBL Adresse

```
oldvec%=BIOS(5,28,L:-1) !alter VBL Vector
```

- ruft die Routine auf, z.B. mit: RCALL chcoll%,reg%()
- dann bekommt man in reg%(8) und reg%(9) die Adressen der zu wechselnden Palettenpuffer zurück
- und kopiert die zu wechselnde Paletten in die Puffer

```
BMOVE palette1%,reg%(8),28
BMOVE palette2%,reg%(9),28
```

falls man sie nicht schon zuvor richtig initialisiert (mit den richtigen Werten belegt hat).

4.9.3 Abspielen digitalisierter Klänge mit dem PSG

Diese Routine habe ich nicht selbst geschrieben, sie ist von Kay Römer, der gut optimierten Assembler-Code programmiert, also eine gute Wahl als Autor einer solchen Routine. Diese Routine spielt zweistimmigen YAMAHA-Sound mit etwa 6 kHz Abspielfrequenz ab und benötigt ca. 22% der Rechenzeit auf einem 8 MHz ST.

Was macht sie:

Ein Einsprung in die Routine mit einem Offset von 110 Bytes initialisiert den Soundchip und richtet mit *Xbtimer (XBIOS 31)* den Timer A auf die eigenTliche Abspielroutine. Dann, bei jedem regulären Aufruf (Offset 0), und der Übergabe von Anfangs- und Endadresse des zu spielenden Samples, werden bis zu zwei Samples (digitalisierte Sounds) gespielt.

Wie benutzt man sie:

- Initialisieren, RCALL startadr%+110,reg%()
- Aufrufen, Sample Startadresse in reg%(6), Endadresse in reg%(7)
  RCALL startadr%,reg%()

4.9.4 Abspielen digitalisierter Klänge via DMA - Sound

Diese Routine ist ein zweistimmiger Sample Player, wobei jede Stimme die volle 8 Bit Auflösung hat, da ich die zwei Samples auf den linken und rechten Kanal verteile. Die Abspielfrequenz beträgt 12,5 kHz, verbrauchte Rechenzeit wohl unter 10%.

Was macht sie:

Bei einem Aufruf mit dem Offset 0 wird der DMA - Sound Initialisiert, wobei wiederholt ein Puffer abgespielt wird. Wenn kein Sound mehr abgespielt werden soll, wird der Puffer mit Nullen gefüllt. Per *Xbtimer (XBIOS 31)* wird der Timer A auf die Abspielroutinen gerichtet und so eingestellt (EVENT COUNT), daß er einen Interrupt auslöst, wenn der DMA-Abspieler am Ende des Puffers ist. Die Abspielroutine selbst holt jeweils ein Langwort von einem Sample und ein Langwort vom anderen Sample und schreibt diese um je ein Byte versetzt in den jeweiligen Puffer. Dieser Puffer wird vom DMA - Sound - Subsystem in Stereo abgespielt, wobei die Bytes an den geraden Adressen auf dem linken und an den ungeraden Adressen auf dem rechten Kanal abgespielt werden. Den Ende eines Samples erkennt die Routine, wenn das Sample ein Langwort mit Null enthält.

Wie benutzt man sie:

- initialisieren mit RCALL startadr%,reg%()
- Aufrufen, Sample Startadresse in reg%(6),
  in reg%(5), 0 = linker u. 4 = rechter Kanal dann
  RCALL startadr%+48,reg%()

4.10 Tips zu ARGON 4

Wie man sieht, ist ein komplettes Spiel schon eine komplexe Sache; die Programmierung eines solchen Spiels für Anfänger komplett zu erklären, ist fast unmöglich. Einige mögliche Fragen kann und will ich erst im zweiten Teil eines Buchs erklären, auch wenn ich im zweiten Teil des Buchs nicht mehr direkt auf Argon 4 eingehe, sind viele Verfahren erklärt, die ich bei der Programmierung des Spiel gebraucht habe. Vieles im Programm ist auf Geschwindigkeit optimiert, aus diesem Grund benutze ich nur globale Variablen, weil der Zugriff auf diese in GFA - BASIC 3.X einfach schneller ist als auf locale und ich dann den Compiler so einstellen kann, daß das erstellte Programm die Prozeduren direkt als Maschinenroutinen anspringt, was wiederum zu schnellerem Ablauf führt. Letztendlich ist in diesem Fall auch der Forschergeist des Lesenden gefragt; einfach ein Backup des Argon 4 Listings machen und damit herumspielen, das ist sicherlich interessanter als alles nachlesen zu müssen.

A.5. Einführung in die Assemblersprache der 680X0 Prozessoren

Einleitung

Obwohl heutige Homecomputer schon "sehr schnell" sind, reicht oft die Geschwindigkeit von Hochsprachen wie C, Pascal und Basic nicht mehr aus, um komplexe Spiele mit riesigen Sprites oder rechenintensive Simulationen zu schreiben. Je höher der Standard bei den Spielen wird, desto tiefer müssen wir, die Programmierer, bei den "kleineren" Geräten auf die System - Ebene herunter steigen um die Leistung, die wir für dessen Realisation benötigen, dem Rechner zu "entlocken".

Was ist Assembler ?

Assemblercode ist die dem Menschen "verständliche" Form der Maschinensprache, wobei der Assembler ein Programm ist, der diese Sprache in die vom Prozessor verständliche Form umwandelt. Danach wird dieser sogenannte Objekt Code von einem Linker zu einem ausführbaren Programm gemacht.
Dem Anfänger empfehle ich den TurboAss von Sigma - Soft als Assembler zu nutzen, denn er ist ein Shareware Programm und in diversen PD-Pools erhältlich, wenn er ihnen jedoch gefällt, sollten sie sich fairerweise registrieren lassen !!!
Der Assembler von Sigma - Soft mahnt die Syntax - Fehler schon beim Schreiben, was ihn für Anfänger besonders wertvoll macht und viel Zeit bei der Fehlersuche erspart.

Die 680X0 Prozessoren

Zeittafel

	1979 -	68000
	1983 -	68010
	1985 -	68020
	1987 -	68030
	1989 -	68040
etwa	1994 -	68060

5.1 Aufbau des Prozessors *(ein "bißchen" zum Einstieg)*

5.1.1 Die Daten/Adressregister

Die Prozessoren der 68000 Familie besitzen 8 Daten **(D0 - D7)** und 8 Adressregister **(A0 - A7)**. Die Datenregister kann man sich als Integervariablen vorstellen, sie sind 32 Bit breit, d.h. ein komplettes Langwort paßt in je ein Register. Das ist schon beim MC 68000 so, obwohl er "nur" einen 16 Bit breiten Datenbus hat, d.h. er muß, um ein Register zu füllen zweimal auf den Speicher zugreifen, denn die Busbreite gibt an, wieviel Bit der Prozessor bei einem (!) Zugriff auf den Speicher erhält. Alle Prozessoren ab dem 68020 haben einen 32 Bit Datenbus, sie können auf 16 Bit genauso schnell zugreifen wie auf 32 Bit. Die Adressregister wiederum kann man sich als "Zeiger" auf einen bestimmten Speicherbereich vorstellen. Auch sie müssen 32 Bit breit sein, damit sie auf jede beliebige Adresse im Speicher zeigen können. Zwar kann der 68000 nur Adressen bis zu einer Größe von 24 Bit (ca. 16.7 Millionen) verwalten, aber ab dem 68020 kann die 680X0 Familie mehrere Gigabytes verwalten.
Das richtige Nutzen von Registern kann einen großen Gewinn an Rechengeschwindigkeit bedeuten, da der Prozessor die für die Befehle benötigten Daten nicht mehr aus dem Speicher "nachladen" muß, und jeder Zugriff auf den Speicher braucht Zeit.

A.5. Einführung in die Assemblersprache der 680X0 Prozessoren

Einleitung

Obwohl heutige Homecomputer schon "sehr schnell" sind, reicht oft die Geschwindigkeit von Hochsprachen wie C, Pascal und Basic nicht mehr aus, um komplexe Spiele mit riesigen Sprites oder rechenintensive Simulationen zu schreiben. Je höher der Standard bei den Spielen wird, desto tiefer müssen wir, die Programmierer, bei den "kleineren" Geräten auf die System - Ebene herunter steigen um die Leistung, die wir für dessen Realisation benötigen, dem Rechner zu "entlocken".

Was ist Assembler ?

Assemblercode ist die dem Menschen "verständliche" Form der Maschinensprache, wobei der Assembler ein Programm ist, der diese Sprache in die vom Prozessor verständliche Form umwandelt. Danach wird dieser sogenannte Objekt Code von einem Linker zu einem ausführbaren Programm gemacht.
Dem Anfänger empfehle ich den TurboAss von Sigma - Soft als Assembler zu nutzen, denn er ist ein Shareware Programm und in diversen PD-Pools erhältlich, wenn er ihnen jedoch gefällt, sollten sie sich fairerweise registrieren lassen !!!
Der Assembler von Sigma - Soft mahnt die Syntax - Fehler schon beim Schreiben, was ihn für Anfänger besonders wertvoll macht und viel Zeit bei der Fehlersuche erspart.

Die 680X0 Prozessoren

Zeittafel

	1979 -	68000
	1983 -	68010
	1985 -	68020
	1987 -	68030
	1989 -	68040
etwa	1994 -	68060

5.1 Aufbau des Prozessors *(ein "bißchen" zum Einstieg)*

5.1.1 Die Daten/Adressregister

Die Prozessoren der 68000 Familie besitzen 8 Daten **(D0 - D7)** und 8 Adressregister **(A0 - A7)**. Die Datenregister kann man sich als Integervariablen vorstellen, sie sind 32 Bit breit, d.h. ein komplettes Langwort paßt in je ein Register. Das ist schon beim MC 68000 so, obwohl er "nur" einen 16 Bit breiten Datenbus hat, d.h. er muß, um ein Register zu füllen zweimal auf den Speicher zugreifen, denn die Busbreite gibt an, wieviel Bit der Prozessor bei einem (!) Zugriff auf den Speicher erhält. Alle Prozessoren ab dem 68020 haben einen 32 Bit Datenbus, sie können auf 16 Bit genauso schnell zugreifen wie auf 32 Bit. Die Adressregister wiederum kann man sich als "Zeiger" auf einen bestimmten Speicherbereich vorstellen. Auch sie müssen 32 Bit breit sein, damit sie auf jede beliebige Adresse im Speicher zeigen können. Zwar kann der 68000 nur Adressen bis zu einer Größe von 24 Bit (ca. 16.7 Millionen) verwalten, aber ab dem 68020 kann die 680X0 Familie mehrere Gigabytes verwalten.
Das richtige Nutzen von Registern kann einen großen Gewinn an Rechengeschwindigkeit bedeuten, da der Prozessor die für die Befehle benötigten Daten nicht mehr aus dem Speicher "nachladen" muß, und jeder Zugriff auf den Speicher braucht Zeit.

5.1.2.1 Das Adressregister A7 - der Userstack

Das achte Adressregister ist etwas ganz besonderes, man nennt es Stackpointer d.h. einen Zeiger auf den Stack. Der Stack oder Atic ist ein Stapelspeicher. Der Stack wächst von oben nach unten und der Stackpointer **(A7)** zeigt (wenn alles gut geht) immer auf das momentane Ende des Stacks. Man könnte theoretisch jedes beliebige Adressregister als Stack benutzen, allerdings gibt es bestimmte Befehle, die nur auf den Stack zugreifen, die in ihrer Ausführungszeit optimiert sind. Zudem wird auf dem Stackpointer die Rücksprungadresse, bei einem möglichen Aufruf einer Unterroutine, abgelegt, das ist so ähnlich wie der Aufruf einer Procedure in GFA - Basic.

5.1.2.2 Das Adressregister A7' - der Supervisorstack

Zuerst muß ich mal erklären, was der Supervisormode ist. Der Supervisormode ist die Betriebsart, in der das Programm auf das komplette System zugreifen darf, d.h. alle möglichen Speicherbereiche schreiben und lesen darf. Zudem kann man bestimmte Befehle nur im Supervisormodus ausführen. Normalerweise befindet sich der Atari im Usermode, in dieser Betriebssart sind bestimmte "lebenswichtige" Speicherbereiche vom Programm (selbst bei einem Fehler) nicht zu erreichen, das bedeutet einen relativ sicheren Schutz vor einem Systemabsturz. Zu den Aufgaben, die der Stack im Usermode hat, wird er im Supervisormode, bei Interrupts, als Ablage vom Status des Prozessors gebraucht. Bei Prozessoren ab 68020 gibt sogar zwei Supervisorstackpointer **ISP** (Interruptstackpointer) und **MSP** (Masterstackpointer). Der ISP hat dieselben Aufgaben wie der **SSP** (Supervisorstackpointer) beim 68000, der **MSP** ist eigentlich gedacht, um dem Betriebssystem einen eigenen Stack zu erlauben, der sich auch während eines Interrupts nicht ändert.

Achtung ! der Stackpointer (sowohl **USP**, **SSP**,**ISP** als auch **MSP**) zeigt immer auf eine gerade Adresse (das unterste Bit wird nicht ins Register übernommen), das ist der Hauptunterschied zu den anderen Adressregistern.

5.1.3 Das Statusregister

Das Statusregister ist ein 16 Bit breites Register und besteht aus zwei Teilen. Der obere Teil besteht aus den Trace Bits, dem Supervisor Bit, Master/Interrupt Status Bit und der Interrupt Maske, läßt sich nur im Supervisormode des Prozessors manipulieren, dieser Teil wird System-Byte genannt; mit ihm werde ich mich noch beschäftigen. Der untere Teil, das Condition Code Registe **CCR** oder User-Byte, kann in allen Prozessorbetriebsarten beschrieben und gelesen werden. Das CCR besteht aus dem Extented, Negativ, Zero, Overflow und dem Carry-Flag. Diese Flags sind Bits, die beim Eintreten bestimmter Situationen vom Prozessor gesetzt werden.

Diese Flags werden z.B. gesetzt, wenn:

eXtented Flag	ähnlich Carry Flag, wird aber bei bestimmten Befehlen z.B. addx, subx, rox ... bei der Ausführung des Befehls berücksichtigt.
Negativ Flag	wenn bei einer Subtraktion das Ergebnis negativ wird.
Zero Flag	wenn beim Bewegen eines Werts vom Speicher in das Register der übertragene Wert Null ist.
Overflow Flag	wenn bei einer Division der Dividend kleiner als der Divisor ist.
Carry Flag	wenn bei einer Addition die Größe des Zielregisters überschritten wird.

Das waren die für den Einsteiger wichtigsten Register, kommen wir nun zu den Adressierungsarten.

5.2. Die Adressierungsarten des 68000

(die zusätzlichen Adressierungsarten der Prozessoren ab 68020 behandele ich erst am Ende des Kapitels)

5.2.1 Register direkt

"Register direkt" bedeutet direkt in ein Register (gehe nicht über Los und ziehe nicht 4000 Mark ein).

```
MOVE.L A1,A2        oder
MOVE.L D2,D1        oder
MOVE.W D1,A5
```

ist Register direkt ! Es wird ein Wert direkt von einem zu einem anderen Register übertragen.

Auch

```
CLR.L D0
```

ist Register direkt, denn das angegebene Register wird gelöscht (*CLR* Abkürzung für CLeaR). Wenn es ein "direkt" gibt, gibt es auch ein indirekt !

5.2.2 Adressregister indirekt

(Register Indirect Addressing)

```
MOVE.L (A0),D0
```

bewege das, was in der Adresse steht (**.L** steht für Langwort und bewegt volle 32 Bit), auf die das Register '**A0**' zeigt in Register '**D0**'

- also "durch die Brust ins Knie".

- mit Postinkrement (Erhöhung danach)

MOVE.L (A0)+,D0

ähnlich wie oben, jedoch wird nach der Operation der Inhalt von 'A0' um

- <Befehl>.L	4 Bytes	-	Befehl benutzt	32 Bit Daten
- <Befehl>.W	2 Bytes	-	" " " "	16 Bit Daten
- <Befehl>.B	1 Byte	-	" " " "	8 Bit Daten

weitergezählt, also um genau so viele Bytes, wie wir übertragen haben, das ist ideal zum Kopieren von Speicherbereichen (wenn man ausreichend Register frei hat, ist ein *MOVEM* zum Kopieren von Speicherbereichen schneller! Dazu später mehr).

wo es ein Plus gibt, gibt es auch ein Minus

- mit Predekrement (erniedrigen davor)

MOVE.L -(A0),D0

zieht je nach übertragenen Byte(s) (**.B .W .L**), ein, zwei oder vier Byte(s), vor der Ausführung des MOVE Befehls vom Adressregister ab.

"Also das Ganze nur andersherum !!!"

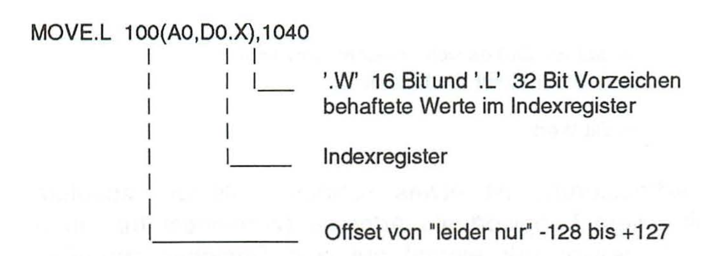
- mit Adressdistanz

```
MOVE.L -200(A0),D0
        I
        I____ Offset, darf nicht größer werden wie 16 Bit,
              signed Integer (+32768 bis -32767)
```

die Adresse, von der in diesem Fall übertragen würde, errechnet sich aus dem Inhalt des Adressregisters+Offset. Nehmen wir an, der Inhalt von **A0** wäre '2000', so würde der Prozessor dann '200' von '**A0**' abziehen also '2000-200' -> also '1800' und würde den Inhalt dieser Adqresse in das Adressregister 'D0' kopieren.

Das geht natürlich noch etwas "ausgefeilter" !

- mit Adressdistanz und Index



MOVE.L 100(A0,D0.X),1040

5.2.3 Absolute Adressierung

- Absolut lang
(Absolute Data Addressing Long)

```
MOVE.L 520,700000
       I     I
       I_____I_____ 32 Bit Wert
```

dieser Befehl überträgt das an der Adresse '520' stehende Langwort, also die Bytes von 520 bis 524, an die Adressen 1040 bis 1044. Diese Art der Adressierung ist "1. Eine Speicherverschwendung", da der Befehl zusätzlich zwei Langworte enthalten muß und daraus resultiert "2. Seine geringe Ausführungsgeschwindigkeit", da der Prozessor beim Einladen der vielen Wörter (Befehlswort und Daten bzw. Adressen) auf den Speicher zugreifen muß und das ist für den Prozessor immer die Operation, für die er "extrem" lange braucht (abgesehen von einer Multiplikation). Allerdings ist diese Operation, beim Übertragen einzelner (nicht zusammenhängender) Inhalte von Adressen, schneller als wenn man jede Adresse erst in ein Register und dann das Register indirekt kopiert.

- Absolut kurz

(Absolute Data Addressing short)

```
MOVE.L d1,128.w
         I     I
         I     I____  .w gibt an, daß es sich um einen vorzeichen-
         I            behafteten 16 Bit Wert handelt
         I
         I__________  16 Bit Wert
```

Diese Art der Adressierung ist etwas schneller als die absolute Adressierung mit einem Langwort als Adresse (zumindest für einen 68000), da der Prozessor nur einmal auf den Speicher zugreifen muß, um die Adresse zu laden und nicht zweimal.

5.2.4 Konstanten Adressierung

(Immediate Addressing)

```
MOVE.L #12345678,D0
       I
       I_____ Wichtig ! den sonst wird die nachfolgende Zahl
              vom Assembler als Adresse und nicht als Konstante
              Interpretiert.
```

bewegt die Zahl '12345678' in das Datenregister '**D0**'

oder

```
move.b #'K',D0
```

bewegt den ASCII-Code von "K" in D0, wobei die Hochkommas anzeigen, daß es sich um ein Zeichen handelt.

5.2.5 PC relative Adressierung

(Program Counter Relative Addressing)

'**PC**' ist die Abkürzung für Program Counter, das ist sozusagen ein Zähler im Prozessor, damit dieser auch immer weiß, wo er sich gerade im Programm (also im Speicher) befindet.

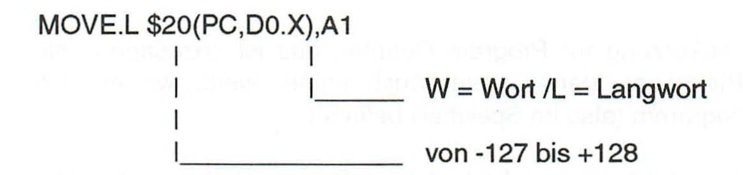
Die PC relative Adressierung bedeutet im Prinzip, daß man bei Befehlen nicht absolute Adressen angibt sondern nur noch die Distanz zu der gewünschten Adresse. Damit ist es möglich, sogenannten "position independent code" zu erzeugen, was nicht mehr und nicht weniger ist als ein Programm, das, ohne daß man seine Adressen an den jeweilgen Speicherbereich anpaßt (das macht normalerwiese das Betriebssystem mit der vor dem Programm befindlichen Reloziertabelle), gestartet werden kann, was wiederum bedeutet, daß man es auch mal eben in einen anderen Speicherbereich kopieren und starten kann. Die ganzen Programme, die in GFA - Basic als INLINE laufen (siehe auch Argon 4) müssen z.B. PC relativ geschrieben sein, da GFA - Basic beim Start eines Programms die INLINES nicht relozieren kann. Man kann natürlich versuchen, das selbst zu übernehmen. Ein weiteres Beispiel für PC relative Programme sind Programme, die im Bootsektor stehen z.B. die Nachricht eines Virenschutzes im Bootsektor oder eben die Viren selbst. Diese müssen relativ sein, da z.B. nicht genügend Platz für eine Reloziertabelle im Bootsektor ist und niemand genau weiß, wohin der Bootsektor im Rechner XYZ geladen wird.

- PC relativ

```
MOVE.L 3200(PC),D0
```

nehmen "wir" an, das Programm würde an der Adresse 5000 liegen, dann würde in unserem Beispiel der Inhalt von der Adresse 8200 in das Register **D0** kopiert werden.

- PC relativ indiziert mit Adreßdistanz



Ähnlich wie Adressregister indirekt mit Adressdistanz und Index, allerdings wird statt einem Adressregister der Programmcounter als Basis für die Berechnung der Adresse genutzt.

5.2.5 PC relative Adressierung

(Program Counter Relative Addressing)

'**PC**' ist die Abkürzung für Program Counter, das ist sozusagen ein Zähler im Prozessor, damit dieser auch immer weiß, wo er sich gerade im Programm (also im Speicher) befindet.

Die PC relative Adressierung bedeutet im Prinzip, daß man bei Befehlen nicht absolute Adressen angibt sondern nur noch die Distanz zu der gewünschten Adresse. Damit ist es möglich, sogenannten "position independent code" zu erzeugen, was nicht mehr und nicht weniger ist als ein Programm, das, ohne daß man seine Adressen an den jeweilgen Speicherbereich anpaßt (das macht normalerwiese das Betriebssystem mit der vor dem Programm befindlichen Reloziertabelle), gestartet werden kann, was wiederum bedeutet, daß man es auch mal eben in einen anderen Speicherbereich kopieren und starten kann. Die ganzen Programme, die in GFA - Basic als INLINE laufen (siehe auch Argon 4) müssen z.B. PC relativ geschrieben sein, da GFA - Basic beim Start eines Programms die INLINES nicht relozieren kann. Man kann natürlich versuchen, das selbst zu übernehmen. Ein weiteres Beispiel für PC relative Programme sind Programme, die im Bootsektor stehen z.B. die Nachricht eines Virenschutzes im Bootsektor oder eben die Viren selbst. Diese müssen relativ sein, da z.B. nicht genügend Platz für eine Reloziertabelle im Bootsektor ist und niemand genau weiß, wohin der Bootsektor im Rechner XYZ geladen wird.

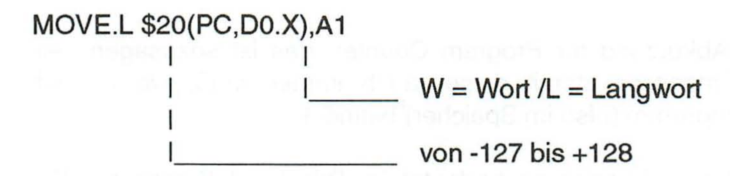
- PC relativ

```
MOVE.L 3200(PC),D0
```

nehmen "wir" an, das Programm würde an der Adresse 5000 liegen, dann würde in unserem Beispiel der Inhalt von der Adresse 8200 in das Register **D0** kopiert werden.

- PC relativ indiziert mit Adreßdistanz

MOVE.L $20(PC,D0.X),A1



Ähnlich wie Adressregister indirekt mit Adressdistanz und Index, allerdings wird statt einem Adressregister der Programmcounter als Basis für die Berechnung der Adresse genutzt.

5.3 Der Einstieg in die Programmierung

Genug der grauen Theorie !

Jetzt etwas Praxis !

5.3.1 Aufruf einer Betriebssystem Routine

Das Betriebssytem TOS (Tramiel Operating System - "Tramiel's Betriebssystem"), möchte in vielen Fällen die Übergabewerte auf den Stack übertragen haben, an dieser Stelle zeige ich Ihnen, wie das geht.

Zuvor ein paar Regeln zur Syntax

hinter einem

' ; '	oder '*' folgt ein Kommentar, wird vom Assembler ignoriert
' # '	folgt eine Konstante
' ' '	z.B. 'A', ein so "geklammertes" Zeichen wird in einen ASCII-Wert gewandelt
'NAME:'	ist ein Label und wird vom Assembler in eine absolute Adresse gewandelt, kann auch PC-relativ verwendet werden.

Es ist möglich, in Assembler z.B. "Worten" bestimmte Werte zuzuweisen,

z.B. ZWEI EQU 2

oder, je nach Assembler ZWEI = 2

Lesen Sie dazu bitte entsprechendes Kapitel in Ihrem Assemblerhandbuch.

****** Beispiel ****** (ASM1.S in KAPA5.TOS auf der Diskette)

; Programm zum Ausgeben eines Zeichens

```
GEMDOS   EQU 1
CONOUT   EQU 2

    MOVE.W  #'A',-(SP)          ; legt den ASCII-Wert von A auf den
                                ; Stack

    MOVE.W  #CONOUT,-(SP)       ; schreibt die Funktionsnummer auf
                                ; den Stack

    TRAP    #GEMDOS             ; ruft das Betriebssystem
    ADDQ.l  #4,SP               ; setzt den Stackpointer wieder
                                ; auf seinen alten Wert

; Raus aus dem Programm (QUIT)

    CLR.W   -(SP)               ; übergibt 0 auf den Stack
                                ; entspricht 'MOVE.W #0,-(SP)'
    TRAP    #GEMDOS             ; ruft das GEMDOS

.END
```

Gehen wir mal genauer auf das kleine Programm ein,

MOVE.W #'A',-(SP)

es zieht zuerst 2 Bytes vom Adressregister 'A7' (auch 'SP' also Stackpointer) ab und legt den ASCII-Wert von 'A' (das umwandeln erledigt der Assembler) in die Adresse, auf die 'A7', also der Stackpointer (auch 'SP') zeigt, ab. Siehe dazu auch Adressregister "Indirekt mit Predekrement" und Konstanten-Adressierung.

Der nächste Befehl

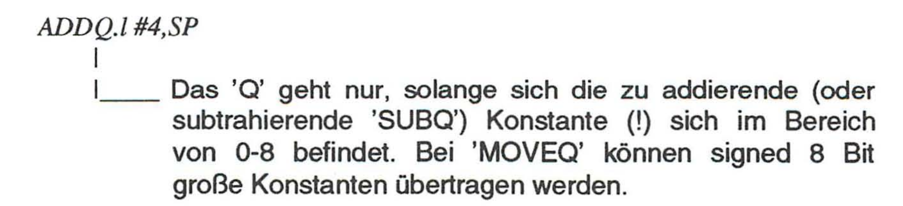
MOVE.W #CONOUT,-(SP)

legt in gleicher Weise die Funktionsnummer der Betriebssystemroutine auf den Stack (der Bereich, auf den der Stackpointer zeigt).

Der

TRAP #GEMDOS

ist der Aufruf der Exception (Ausnahme). Beim MC68000 liegt in den unteren 256 Bytes des adressierbaren Speichers eine Vektortabelle, bei einem Interrupt (Unterbrechung) wechselt der Prozessor in eine Ausnahmeverarbeitung und springt an die Adresse, die an der für diesen Trap vorgesehenen Adresse eingetragen wurde (z.B. durch das Betriebssystem). An dieser Adresse befindet sich der sogenannte Trap Dispatcher (Dispatcher = Verteiler), dieser wiederum liest die Funktionsnummer vom Stack und verzweigt in die entsprechende Betriebssystemroutine. Die wiederum nimmt (kopiert) die für sie bestimmten Daten vom Stack und verrichtet ihre Arbeit, in unserem Fall kopiert sie die Bitmap des Buchstaben 'A' auf den Bildschirm.
Um den Stackpointer, den wir in seiner Lage verschoben haben, wieder auf seine ursprüngliche Position zu bringen, addieren wir die Menge der auf den Stack plazierten Bytes, um den wir ihn nach unten verschoben haben, wieder mit



auf ihn drauf.

5.4 Das Umschalten in den Supervisor Mode

Wenn man sich im User-Mode befindet, ist die Welt noch in Ordnung, Systemabstürze kommen nur gelegentlich vor und auf die gefährlichen Speicherbereiche ganz weit oben (Hardwareregister) und ganz weit unten (Interrupt-Vektoren, System-Variablen) darf man eben nicht zugreifen !

Aber das soll sich jetzt ändern !!!

Beim Einschalten des Computers oder nach einem Reset befindet sich der Rechner automatisch im Supervisormode; zu diesem Zeitpunkt trägt das Betriebssystem sich in die Interuptvektoren des Betriebssystems ein, und initialisiert die Hardwareregister (d.h. trägt eine Grundeinstellung ein). Dann schaltet das Betriebssystem in den Usermode und jedes dann gestartete Programm befindet sich im Usermode. Jedoch jede Routine, die im Interrupt ausgeführt wird, befindet sich im Supervisormode. Es wäre nun möglich, sich in einen Interruptvektor einzutragen und dann diesen Interrupt zu erzeugen. Im Usermode kann man sich jedoch in keinen Interruptvektor eintragen ! Die einzige Möglichkeit, in den Supervisormode zu wechseln, ist das Betriebssystem, denn das 'hängt' ja schon in einigen Vektoren.

5.4.1 Umschalten in den Supervisor-Mode

Beispiel

```
;Programm zum Umschalten in den Supervisor Mode
;in Kapitel8.TOS Datei ASM2.S
;****** Von User in den Supervisor Mode *******
;
TEXT
GEMDOS EQU 1
SUPER  EQU 20
;
      CLR.L  -(SP)
      MOVE.W  #SUPER,-(SP)    ; Funktionsnummer auf den Stack
      TRAP   #GEMDOS          ; GEMDOS TRAP
      ADDQ.L  #6,SP           ; Stack auf alten Wert
;
;**************************************
;
;In 'D0' befindet sich jetzt der User Stackpointer
;
;****** Rette alten Stackpointer ********
;
      MOVE.L  D0,OLDSTACK(PC) ; User Stackpointer retten
;
;****************************************
;
;
;Ab jetzt Supervisormode !
;
;Hier kann man jetzt z.B. Auf die Hardwareregister zugreifen
;oder die Systemvektoren
;
;
```

```
;******* Zurück in den Usermode ********
;
        MOVE.L  OLDSTACK(PC),-(SP)      ; Alten Stackpointer
                                        ; auf Stack
        MOVE.W  #SUPER,-(SP)            ; Funktionsnummer auf Stack
        TRAP    #GEMDOS                 ; GEMDOS TRAP
        ADDQ.L  #6,SP                   ; Stackpointer auf alte
                                        ; Position
;
;*****************************************
;
;
;     ******* QUIT ******
;
        CLR.W   -(SP)           ; Funktionsnummer auf Stack
        TRAP    #GEMDOS         ; GEMDOS TRAP
;
DATA  ; Gibt an, daß jetzt nur noch Daten kommen, wird
      ; vom Betriebssystem angelegt und nicht mit dem
      ; Programm abgespeichert
;
OLDSTACK: DS.L 1        ; Ein Langwort für alten Stackpointer
                        ; reservieren
```

Sofern das Programm komplett im Supervisor Mode ablaufen soll, ist diese Art der Umschaltung sinnvoll, auch Betriebssystemroutinen lassen sich immer noch aufrufen.

5.4.2 Ausführen einzelner Routinen im Supervisor-Mode

Wenn man nun nur einzelne Programmteile im Supervisor Mode ausführen lassen möchte, kann man das mit dem *XBIOS 38* Befehl Superexec tun. Allerdings lassen sich in diesem Programmteil dann ohne weiteres keine Betriebssystemfunktionen mehr aufrufen.

Beispiel:

```
;Programm, das Routinen im Supervisor-Mode aufruft
;über XBIOS 38 - Superexec
;
;Kapitel5.TOS ASM3.S
;
SUPEREXEC EQU 26        ; Funktionsnummer
XBIOS     EQU 14        ; extented basis input output - Xbios Trap
;
;***** Aufruf von Superexec *****
;
        PEA     ROUTINE             ;Adresse der auszuführenden
                                    ;Routine auf den Stack
        MOVE.W #SUPEREXEC,-(SP)     ;Funktionsnummer a.d. Stack
        TRAP    #XBIOS              ;XBIOS Betriebssystem
                                    ;Routinen aufrufen
        ADDQ.L #6,SP                ;Stack zurück
;
;   ****** QUIT ********* siehe oben !!
;
        CLR.W  -(SP)
        TRAP    #1
;
;
```

```
;****** Hier folgt aufzurufende Routine *****
;
ROUTINE:            ;Label der anzuspringenden Routine
;
; Programmcode im Supervisor-Mode
; Jedoch keine Betriebssystem-Aufrufe !!!
;
        RTS         ; Wird mit einem RTS d.h.
                    ; return from suproutine beendet
                    ; "springt" dann hinter den TRAP #XBIOS
END
```

8.5 Schleifen und Entscheidungsanweisungen in Assembler

Auf der Diskette befinden sich vollständige und lauffähige Programme, bei den kommenden Beschreibungen werde ich (auch der Übersicht halber) schon erklärte Programmteile einfach weglassen und mich auf das "Wichtigste" konzentrieren.

```
;***** ähnlich FOR NEXT *****
;KAPITEL5.TOS
;ASM4.s
;
        MOVE.W  #100,D0
SCHLEIFE:
;
; Alles was hier kommt wird 101 mal ausgeführt
;
        DBRA D0,SCHLEIFE
;
;******** QUIT ******* übertrage von oben
END
```

Dieses Programm wäre vergleichbar mit dem BASIC Programm

```
For T&=0 to 100
  'Programmteil
NEXT T&
```

```
;***** ähnlich  Repeat Until
;KAPITEL5.TOS
;ASM5.S
;
REPEAT: ;Label
;
; Hier würde jetzt ein Programmteil kommen
;
        CMP.W #100,D0     ; CMP CoMPare - vergleiche 100 mit
                          ; dem Inhalt von D0
        BNE.S REPEAT      ; Branch if Not Equal; verzweige, wenn ungleich
******* QUIT **********
END
```

Ist vergleichbar mit

```
REPEAT
  ' Hier würde "etwas Programm" kommen
UNTIL a&<>100
```

Was macht nun der 'CMP' Befehl ?

Der '**CMP**' setzt die condition codes im Statusregister ! Je nach dem, welche condition codes gesetzt sind, reagiert z.B. der bedingte Sprungbefehl anders.

Tabelle - Wirkung des CMP Befehls aus dem Status Byte

CONDITON CODE	BEDEUTUNG	Status Byte
CC	Carry Clear	C=0
CS	Carry Set	C=1
EQ	EQual	Z=1
F	False	egal, immer erfüllt
GE	Greater or Equal	N=1, Z=1
GT	Greater Than	N=1
HI	HIgher	C=1
HS	Higher or Same	C=1 oder Z=1
LE	Less or Equal	Z=1
LO	LOwer	alle 0
LS	Less or Same	alle 0 oder Z=1
LT	Less Than	Z=1
MI	Minus	N=1
NE	Not Equal	Z=0
PL	Plus	N=0; Z=1
T	True	egal, immer erfüllt
VC	oVerflow Clear	V=0
VS	oVerflow Set	V=1

CMP bildet die Differenz zwischen Ziel und Quelle und setzt je nach Ergebnis das Status-Byte z.B. in 'D0' befindet sich der Wert 100, dieser wird mittels 'CMP #99,D0' mit 99 verglichen, dann setzt der Prozessor das Negativ Flag, weil 99-100 kleiner als Null ist. Das Setzen der Flags im Status Byte erfolgt nicht nur durch die Vergleichsoperationen sondern im Prinzip bei allen Prozessoroperationen, allerdings sollte man darauf achten, daß jeder Befehl nur bestimmte Flags setzen kann. Informationen darüber erhält man durch Probieren mit einem guten Debugger oder durch Nachlesen in Prozessorbüchern (siehe Literaturverzeichnis).

5.6 68030-Spezifisches

5.6.1 Aus/Einschalten der 68030 Caches

Eine weit verbreitete "Unart" guter Demo Programmierer ist das Schreiben von selbstmodifizierendem Code. Selbstmodifizierend ist ein Programm, wenn es sein eigenes Textsegment ändert, d.h. Teile des eigenen Programmtextes mit einem neuen "Programm" überschreibt. Meist sind es nur Conditions oder Konstanten, die im bestehenden Programm geändert werden, das geht meist schneller als das Verzweigen in andere Routinen. In Situationen, wo der Demo/Spieleprogrammierer an die Grenzen der Prozessorleistung stößt (wenn wirklich nichts mehr zu optimieren geht) greift er gern zu dem Mittel, kleine Änderungen direkt im Programm vom Programm selbst vornehmen zu lassen. Auch spart er damit Speicher.
Nun, auf dem 68000 ist diese Technik meiner Meinung nach vollkommen legitim, denn nur so sind viele der wirklich beeindruckenden Demos möglich. Allerdings "sollte" man neue Programme so schreiben, daß sie auf allen ATARI-Rechnern laufen! Und daher, wenn möglich, den Prozessor testen (siehe Kapitel über die Kom patibilität) und zumindest an den kritischen Stellen auf Geräten mit Prozessoren > 68020 den Cache ausschalten, besser noch eine alternative Routine bieten, die die Fähigkeiten des jeweiligen Prozessors nutzt. Auch sollte man versuchen, Cache-Beschleuniger-Karten oder MEGA STE's zu berücksichtigen, d.h. den zu modifizierenden Teil des Programms nicht weiter als 16 KB von dem zu modifizierenden Teil weg zu plazieren, damit die ganze Sache komplett im Cache stattfindet.

Zum Aus/Einschalten der Caches muß

```
MOVEC CACR,D0        ; DC.W $4e7a,2
ANDI #$FEFE,D0       ; ORI #$3111,D0 zum Einschalten
MOVEC D0,CACR        ; DC.W $4e7b,2
```

im Supervisor-Mode ausgeführt werden.

*** Achtung ! *** das Desktop schaltet nach Beendigung eines Programms die Caches, je nach Voreinstellung, wieder an oder aus. (bei TOS 3.01)

Der '*MOVEC*' Befehl ist ab dem 68010 in der Familie der 680X0 Prozessoren integriert. Er bietet Zugriff auf die neuen Register dieser Prozessoren und zwar die SFC (Source Function Code) und die **DFC** (Destination Function Code) ab 68010, diese beiden Register sind je 3 Bit breit und setzen drei Pins (die an allen 68000 Prozessoren vorhanden sind), je nach gesetzten oder gelöschten Bits, hoch bzw. niederohmig. Damit sind spezielle hardwareorientierte Speicherkonzepte möglich (eine Erklärung würde zu weit führen, da dies zur Systemprogrammierung gehört). Zudem läßt sich noch das **VBR** (Vector-Basis-Register) durch den 'MOVEC' verändern, was eine Verschiebung der Vektortabelle des Prozessors zu Folge hat. Damit ist es z.B. möglich, mehrere alternative Vektortabellen gleichzeitig im Speicher zu halten, das wiederum vermeidet lästiges Umkopieren (Systemprogrammierung). Ab dem 68020 kann man mit dem '*MOVEC*' Befehl noch die Register zu Cacheverwaltung 'CAAR' (CAche Address Register) und 'CACR' (CAche Control Register) manipulieren. Für uns dürfte allerdings nur das 'CACR' Register interessant sein, da man mit ihm den Cache ein- und ausschalten kann. Alle anderen Funktionen (außer beim Einschalten gleich den Burst-Mode zu aktivieren) dürften uninteressant sein, da beim 68040 (z.B. im Falcon 40) diese auf andere Register ausgelagert oder erst gar nicht implementiert (die Kunst liegt im weglassen) worden sind. Näheres dazu finden Sie in jedem besseren Prozessorbuch (siehe Literaturverzeichnis).

'ANDI'/'ORI' sind Befehle, die eine logische UND- bzw. ODER-Verknüpfung einer Konstante mit einem Ziel ausführen (in unserem Fall einem Register). Da beim '**CACR**' Register eine Menge von Bits manipuliert werden können, empfiehlt es sich, nur die Bits zu setzen bzw. zu löschen, die man wirklich setzen/löschen will, dafür sind logische Verknüpfungen bestens geeignet. Direktes Beschreiben dieses Registers ist nicht zu empfehlen, da noch unklar ist, welche Wirkungen die reservierten Bits in späteren Prozessorversionen haben werden.

CACR 68020/30/40

31	30	…	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DE 040	0		IE 040	0	WA 030	DBE 030	CD 030	CED 030	FD 030	ED 030	0	0	0	IBE 030	CI 020/030	CEI 020/030	FI 020/030	EI 020/030

nur 68040

DE = Enable Data Cache, ein/ausschalten des Daten Caches
IE = Enable Instruction Cache, ein/ausschalten des Befehls Cache

nur 68030

WA = Write Allocate, Cache update ein/ausschalten
DBE = Data Burst Enable, Füllen des Daten Caches im Burst mode
CDE = Clear Data Cache, Löschen des Daten Cache
FD = Freeze Data Cache, Inhalt des Daten Caches einfrieren
ED = Enable Data Cache, Ein/Ausschalten des Daten Caches
IBE = Instruction Burst Enable, Füllen des Befehls Cache im Burst mode

nur 68020/030

CI = Clear Instruction Cache, Löschen des Befehls Cache
CLI = Clear Instruction Cache Entry, Löschen eines Eintrags des Befehls Cache
FI = Freeze Instruction Cache, Inhalt des Befehls Cache einfrieren
EI = Enable Instruction Cache, Ein/Ausschalten des Befehls Cache

5.6.2 Zusätzliche Adressierungsarten ab 68020

Bei allen Adressierungsarten mit Index, kann hinter das Indexregister noch einen 'scale factor' setzen. Mit diesem wird dann das Indexregister multipliziert. Diese Erweiterung bietet sich zur Tabellenverarbeitung an.

Beispiel:

```
MOVE.L 100(A0,D0.w*8),1040
```

der 'scale factor' kann 1, 2, 4, 8 betragen

5.7. Schlußwort zu diesem Kapitel

Natürlich bin ich mir bewußt, daß man nach dem Lesen dieses Kapitels noch lange kein Assembler kann ! Trotzdem hoffe ich, daß ich Ihnen einige wichtige Hinweise zum Verständnis von Assembler und somit zum Verständnis der folgenden Kapitel geben konnte. Gerade den Umsteigern von anderen Geräten, die schon etwas Erfahrung mit dem Assembler ihres Gerätes haben, dürfte ich wohl ein ganzes Stück weiter geholfen haben ?!

A.6. Wie schaffe ich es jetzt, ein eigenes Spiel zu schreiben ?

Nun, sie haben sicherlich schon konkrete Vorstellungen, was für ein Spiel sie schreiben wollen und können es in eine der Kategorien einordnen die jetzt folgen. Sie müssen allerdings verstehen, daß ich nur Tips zu jedem Genre geben kann, ich hoffe daß sie Ihnen weiterhelfen können !

6.1 Adventures

6.1.1 Text Adventures (z.B. Infocom Adventures)

Das Wichtigste bei Textadventures ist der Parser! Um einen guten Parser zu schreiben, sollten sie die Sprache, die der Parser verstehen kann (Deutsch, Englisch, Französisch), gut beherrschen, sie sollten die Schlüsselwörter kennen, die eine Sprache ausmacht. Außerdem sollten sie wissen, welche Wörter sie ignorieren können! Auch sollten sie versuchen, Fehler, die bei der Eingabe entstehen können, von vornherein herauszufiltern.

Ein Beispiel für einen Dialog:

Spieler: *Rede mit König*

Im Computer
- In Groß- oder Kleinschrift wandeln. (Filtert Fehler bei Groß- oder Kleinschreibung heraus)

REDE MIT KÖNIG

- Schlüsselwörter suchen (Das ist natürlich nur ein sehr einfaches Beispiel)

REDE KÖNIG

- Ist diese Funktion in diesem Moment möglich d.h. ist der König im Raum etc. ?

Der Computer müßte somit in einen "Gesprächsmodus" schalten und dann entsprechende Antworten auf entsprechende Inhalte in dem kommenden Dialog geben.

Entsprechende Befehle zum Realisieren eines solchen Algorithmus finden sie bei den Stringfunktionen der jeweiligen Programmiersprache.

6.1.2 Grafik/Text Adventures (z.B. The Pawn, Hellowoon)

Diese haben den Vorteil gegenüber reinen Text - Adventures, daß man die Räume nicht so genau beschreiben muß, denn der Spieler hat ja Augen im Kopf. Es bleibt jedoch das Problem der Bilder, da der Spielspaß proportional mit dem Interesse an den Bildern wächst. Außerdem sind die Bilder das einzige, was der Käufer bei Testberichten in den jeweiligen Zeitungen selbst beurteilen kann !

Jetzt noch ein paar technische Tips !

Zwischen Text und Grafik (z.B. mit TIMER B) die Auflösung umschalten (beim ST zwischen 'niedriger'/'mittlerer' Auflösung). Man hat somit mindestens 16 Farben für das Bild und 80 Zeichen pro Zeile für den Text. Man sollte beim Umschalten den Timer B immer ein paar Zeilen vor dem Beginn des Textes auslösen lassen und entsprechend Platz zwischen Grafik und Text lassen, da sich der Timer B Interrupt (wenn man nicht fast alle anderen Interrupts sperrt oder auf speziell modifizierte, eigene Routinen richtet) um einige Zeilen verschiebt - siehe Farbumschaltung im beigefügten Spiel (Kapitel 4).
Diese Umschaltung bringt jedoch auch Kompatibilitätsprobleme mit sich, da die Rechner der nächsten Generation diese oder jene Grafikauflösung vielleicht gar nicht mehr besitzen, auch verbaut man sich damit die Möglichkeit der Ausgabe auf entsprechende Grafikkarten, die für diese Art von Spielen besonders gut geeignet sind. (Siehe auch dazu Kapitel 8 über die Kompatibilität)

6.1.3 Animierte Grafik Adventures (z.B. Maupitti Island)

Die Grafik muß bei diesen Programmen genauso gut (oder besser) wie die der nur Grafik-Adventures sein. Dafür fällt in den meisten Fällen der Parser weg. Die Programme werden über Pulldown - Pop up Menüs, und/oder Tastenkürzel oder Befehlszeilen am unteren Rand gesteuert. In den meisten Fällen kann man Gegenstände auf dem Bildschirm mit der Maus anklicken und dann angeben, was man mit ihnen anfangen möchte. Auch diese Art von Programmen läßt sich so programmieren, das sie auf allen Atari-Rechnern mit allen Grafikkarten laufen.

6.1.4 Actionadventures (z.B. Larry, Monkey Island u.v.m)

Tja, die sind gar nicht so leicht zu schreiben. Das Hauptproblem ist das Erzeugen von mehren Ebenen, die immer ein Stückchen weiter im Hintergrund liegen. Das kann man z.B. durch das Erzeugen von mehreren Masken realisieren, wobei man immer mehr vom Hintergrund beim Spielersprite ausmaskiert (mit einer Maske vom Hintergrund verknüpft). Das ist nicht so speicherintensiv, da jede Maske nur aus einer Bitplane besteht, wie das Kopieren der jeweiligen Ausschnitte der jeweiligen Ebenen, denn dazu müßte man alle Ebenen in der Tiefe (z.B. eine Palme im Vordergrund) mit allen Bitplanes im Speicher bzw. auf der Diskette haben. Das Konstruieren eines Interpreters, wie der von Sierra, ist in etwa fast so schwer wie eine komplett neue Programmiersprache zu entwickeln, doch wenn man einen solchen Interpreter erst mal hat, kann man jede Menge Spiele entwickeln.

6.2. Action Spiele

6.2.1 Ballerspiele (z.B. Wings of death, Xenon II ...)

Sie sind eigentlich öde, deshalb müssen Elemente her, die diese Art von Spielen interessant machen. Ausgeklügelte Extra Waffen, Bonus - Spiele Extra Leben u.v.m. ! Aber das ist nicht alles, gute Grafik, fließende Bewegungen etc. etc. sind ein Muß.

Das ist für den ungeübten Programmierer kaum zu bewältigen. Wie man sieht (Argon 4), kann ein weniger aufwendig gestaltetes Programm großen Spaß bereiten.

Wenn Sie es nicht einmal schaffen, eine komplette Demo mit Scrolltext, hüpfenden Sprites, digitalem Sound, scrollendem Screen und vielen Rasterinterrupts zu schaffen, also eine richtig gute Demo, sollten Sie nicht versuchen, ein professionelles Ballerspiel zu schreiben (sie werden es nicht los !!! außer sie haben eine geniale Idee). Aber gute PD- oder Shareware-Programme sind ja auch nicht zu verachten und man macht vielleicht irgend jemanden eine (zumindest für eine gewisse Zeit) kleine Freude damit.

6.2.2 Ballerspiele mit Vektorgrafiken (z.B. Starglider)

Dazu benötigt man gute Kenntnisse in Vektoralgebra und gute Algorithmen zum Zeichnen von Linien. Natürlich kann man auch die Funktionen zum Zeichnen von Linien des Betriebssystems (VDI) benutzen. Da diese aber relativ langsam sind, dürfte das erst ab Atari TT oder Falcon030 sinnvoll sein (eine gute Beschleunigerkarte oder eine Grafikkarte mit speziellem Grafikprozessor reicht auch schon). Leider würde eine solche Einführung (ich hasse diesen Satz) den Umfang dieses Buches sprengen, weiterführende Literatur finden Sie im Literaturverzeichnis,siehe Anhang B.

6.2.3 Action - Strategiespiel (z.B. Archon I+II,Lemmings)

Bei dieser Art von Spielen ist meist die Idee entscheidend, viele gute Ideen lassen sich schon relativ einfach realisieren. Es entscheidet eben der Spielwitz und die Suchtgefahr eines Spieles. Am interessantesten für den Spieler ist meistens der Zwei-Spieler-Modus, denn kein Programm kann so interessant sein, wie ein zweiter Spieler, und das wird auch noch einige Jahre so bleiben.

6.3 Simulationen

6.3.1 Handels-/Wirtschafts-/Welten-/Städtesimulation
(z.B. Sim City, Populus, Oil Imperium, Rings of Medusa)

Diese Programme lassen sich meist komplett in einer Hochsprache schreiben, es gibt natürlich auch Ausnahmen wie Populus. Ich kenne z.B. den Programmierer von Oil Imperium recht gut (Hallo Simon, ich hoffe, Du hast nichts dagegen, daß ich dich hier erwähne), daher weiß ich, daß Oil Imperium auf dem ST fast komplett in GFA - Basic 3.0 geschrieben worden ist und nur einige wenige Assembler - Routinen als Inlines hinzugefügt worden sind. Beim Programmieren von Simulationen sollte man darauf achten, daß Spielwitz und Nähe zur Realität sich die Waage halten, und daß das Programm auch ohne Genuß einer mehrere hundert Seiten starke Anleitung spielbar ist, denn wer liest sie schon bevor er spielt?

Worte zum Ende des Kapitels:

Ähnlich wie beim "Wort zum Sonntag", noch ein paar Worte zum Ende des Kapitels.
Der Übergang von einer Art von Spielen zu einer anderen ist fließend und wer verzweifelt nach Informationen über z.B. Sportspiele in diesem Kapitel sucht, findet sie versteckt in all den anderen besprochenen Genres. Das Programmieren ist eine Tätigkeit wie das Modellbauen, man sollte schon wissen, was im Endeffekt dabei herauskommen soll. Es ist aber auch - gerade bei der Spieleprogrammierung - ein kreativer Prozeß, so wie das Malen oder das Komponieren. Trotzdem ist es wichtig, sich zu jedem Schritt Notizen zu machen, Programme im Programmtext zu dokumentieren. Sie müssen nicht unbedingt Flußdiagramme und Ablaufpläne etc. zeichnen! Aber modulare Programmierung, d.h. Zerlegen eines Problems in lauter kleine Teilstücke (z.B. Proceduren) die man mit Extremwerten (Minimas und Maximas) testen kann, sollten sie schon praktizieren. Wenn man dann zu jedem Modul schreibt, was es eigentlich macht, können Sie Ihre Programme mit wachsendem Kenntnisstand immer weiter verbessern und vielleicht auch irgendwann einmal Ihr erstes Programm veröffentlichen !!!

7. Techniken der Demoprogrammierung

Eine der wichtigsten Regeln beim Anschauen von Demos ist: Es ist nichts so schwer, wie es aussieht !
Viele "Screens" (Teile einer Demo) sind, so toll sie auch aussehen mögen, weniger eine programmiererische Glanzleistung als eine Frage des Gewußt-Wie. Um das zu erkennen, muß man ein bißchen was über die Grenzen der ST/STE/TT/Falcon Hardware wissen. Einige Grenzen möchte ich hier aufzeigen und mit welchen Tricks man sie beseitigen kann.

7.1 Die Grenzen der Hardware

7.1.2 Der Sound

Beim Falcon030 müssen diese Grenzen sicherlich noch gesteckt werden. Wozu der DSP bei geschickter Programmierung in der Lage ist, ist noch relativ unklar; unglaubliche Soundeffekte warten darauf, durch geschickte Programmierung in 16 Bit Stereo zu Gehör gebracht zu werden.

Der DMA Soundchip des STE/TT ist schon eher eine bekannte Größe, der zwar von der Hardware nur den einen Speicherbereich 8 Bit Stereo abspielen kann (also ein Stereo DMA Kanal) und via Microwire Interface Höhen und Bässe des abgespielten 2 Kanal Sounds einstellen kann. Das ist im Vergleich zum Paula, der im AMIGA eingebaut ist, nicht gerade überzeugend, jedoch muß man bedenken, daß der DMA Kanal im STE/TT fast keine Prozessorzeit schluckt, da er die Daten genau wie der Video-Shifter dann holt, wenn der Prozessor nicht auf den Speicher zugreift. Das ist das starre Bus-Timing in ST/STE/TT; die CPU muß nur in den seltensten Fällen warten, so daß der CPU durch Zugriffe des VIDEO- und (bei STE/TT) AUDIO-Shifters auf den Bus nur etwa 10% seiner Performance "flöten" gehen. Der AMIGA verliert je nach Auflösung und Abspielfrequenz der gespielten Sounds, Blitter, Copper Einsatz schätzungsweise 70% der CPU Performance im CHIP -

RAM. Da also die ATARI User noch ein bißchen Prozessor-Zeit zur Verfügung haben, verschwenden wir sie einfach mit der Emulation des vierstimmigen Sounds (wie, siehe Kapitel B.4). Das klingt dann auch schon recht brauchbar.

Die Grenzen des YAMAHA- oder baugleichem General Instruments-Soundchips liegen noch ein paar Level tiefer. Eigentlich kann dieses Ding nur rechteckige Wellenformen und Rauschen erzeugen. Da konnte der Chip im ATARI 400/800/600XL - 1200XL/130XE etc. noch besseren Sound von sich geben, denn der hatte wenigstens Filter und war vierstimmig (!). Aber selbst im Vergleich zum müden C64 Sounderzeuger kann dieser LOW LEVEL I/O Chip, den man in den ST's findet, mit seinen geringen Soundfähigkeiten nicht mithalten. Wie man trotzdem digitalen Sound aus diesem Ding herauskitzelt, findet man in Kapitel B.4. Im Kapitel A.4 habe ich ja schon gezeigt, wie man selbst in Basic mit viel Rechteck und Rauschen Musik machen kann.

7.1.2 Die Grafik

Wohl aus technischen Gründen liegen die Bitplanes im Grafikspeicher für je sechzehn Pixel hintereinander (außer "True Color" beim Falcon030). So muß man beim Bearbeiten eines Bildpunktes in einem 256 Farbmodus acht Wörter des Bildschirmspeichers manipulieren. Bei einer VGA - Karte oder beim Archimedes reicht für dieselbe Aufgabe der Zugriff auf ein einziges Byte. Die Organisation des ATARI Bildspeichers ergibt besonders beim horizontalen Scrollen Probleme, da ja der Abstand zwischen dem Wort in dem die einen 16 Pixel für eine Plane und den nächsten 16 Pixel für die gleiche Plane Anzahl der Planes mal einem Wort ist. Das dauernde Arbeiten mit diesen Offsets (Abständen) ist sehr knifflig und kostet relativ viel Rechenzeit. Zudem ist noch zu bemerken, daß die ATARI's zwar ergonomische Bildwiederholfrequenzen (60 Hz in Farbe und 71 Hz in Monochrom) zulassen, aber die Menge der Farben bei ST/STE mit 16 aus 512 bzw. 4096 beim STE sein Maximum erreicht hat, beim Uramiga normal 32 aus 4096 und HAM vol-

le 4096. Ach ja, da wir gerade dabei sind, der HAM Modus (das ist vielleicht auch für ein paar ST-User interessant) funktioniert in etwa so:

HAM MODUS: (AMIGA)

6 Bitplanes für einen Bildpunkt,

die Bits 6 & 7 eines Bildpunkts legen fest:

a) Rot, Grün oder Blauteil des letzten Bildpunkts modifizieren

Bit 6 & 7	01 =	blau	modifizieren
	10 =	rot	modifizieren
	11 =	grün	modifizieren

b) die Farbe eines Farbregisters als Bildpunkt

Bit 6 & 7 00 = Farbwert aus einem von 16 Palettenregister

die restlichen vier Bits eines Bildpunkts legen fest:

a) den Wert von der Rot-, Grün oder Blaumodifizierung

b) welches Farbregister dem Bildpunkt zugeordnet wird

Aber jetzt weiter mit meinen geliebten ATARI Rechnern !

7.2. Ein paar Tricks um die Hardware zu überlisten

Wie man mehr als sechzehn Farben auf einen Bildschirm darstellt, erfährt man in Kapitel B.3; auch habe ich das ja schon im letzten Kapitel angerissen. Der Sound wird, wie schon erwähnt, in Kapitel B.4 ausführlich abgehandelt, also bleibt uns nur noch das Problem mit dem Fine - Scrolling.

7.2.1 Schnelles Scrolling auf den ATARI Rechnern

Information Hardwarescrolling der STE Serie und die Syncscroller sind in Kapitel B.1 nachzulesen. Ich beschränke mich in diesem Kapitel mit der Erklärung von Tricks, die das Scrolling auf den Ur - ST's beschleunigen.

7.2.2.1 Vertikales Scrolling

Bei dem Übertragen von vollen vier Planes gibt es nur eine wirklich sinnvolle Lösung, das Kopieren von Speicher, so wie ich das bei Argon 4 mit dem 'BMOVE' aus GFA - Basic gemacht habe. Dieser 'BMOVE' kann sogar überlappende Speicherbereiche kopieren, das ist bei einer eigenen Routine natürlich nicht nötig. Auf der folgenden Seite finden Sie eine Beispielroutine in Assembler:

Hier eine solche Kopierroutine in Assembler !

```
******** Kopieren von n Zeilen eines Lores - Screens ***********

* In A0 befindet sich die Anfangsadresse Quelle
* In A1 die Anfangsadresse Ziel
* D0 Zähler für die zu übertragenden Zeilen

**************** Kopieren von n Zeilen **************

zeile:

*************** eine Zeile ****************

    movem.l (A0)+,D1-D7/A2-A6       ; 12 Longs lesen (Quelle)
    movem.l D1-D7/A2-A6,(A1)        ; schreiben (Ziel)
    lea     48(A1),A1               ; Addiert 52 Bytes zu A1 dazu
    movem.l (A0)+,D1-D7/A2-A6       ; nochmal 13 Longs lesen (Quelle)
    movem.l D1-D7/A2-A6,(A1)        ; schreiben (Ziel)
    lea     48(A1),A1
    movem.l (A0)+,D1-D7/A2-A6       ; nochmal 13 Longs lesen (Quelle)
    movem.l D1-D7/A2-A6,(A1)        ;  schreiben (Ziel)
    lea     48(A1),A1
    movem.l  (A0)+,D1-D4            ;  die letzten 16 Bytes
    movem.l  D1-D4,(A1)
    lea     16(A1),A1

*********************************************

    dbra    D0,zeile

*********************************************************

    END
```

Nun, was macht das Programm ?

Es kopiert die Menge von Zeilen+1, die in 'D0' stehen von 'A0' nach 'A1', ich benutze hierbei ein 'movem' weil dieser bei der Übertragung von mehr als 3 Register in den Speicher schneller ist als ein 'move'. Ich gehe bei der Programmierung von einem 320*200 - 16 Farb-Bild aus; bei größerem oder kleinerem Speicherbedarf für eine Zeile kann die Routine entsprechend abgeändert werden.

Der Unterschied zwischen einem ST und STE im Bezug auf vertikales Scrolling ist das fehlende LOW - Byte der ST - Serie bei den Video-Basisregistern. Dieses LOW - Byte ermöglicht es dem STE, den Bildschirmspeicher wortweise zu verschieben (siehe dazu auch B.1.2.2.1). Trotz alledem läßt sich der Bildschirmspeicher acht zeilenweise verschieben. Denn der Bildschirmspeicher muß duch das fehlende 'LOW - Byte' auf einer 256 Bytes Grenze liegen.

Beispiel:

Adresse:

200000	DEZIMAL
$030D40	HEXADEZIMAL

HIGH	MID	LOW	
3	13	64	DEZIMAL
$03	$0D	$40	SEDEZIMAL bzw. HEXADEZIMAL

diese Adresse würden STE/Falcon/TT ohne weiteres in ihre Videoregister übernehmen (z.B. via setscreen bzw. XBIOS(5)) der ST allerdings würde nur

HIGH	MID
$03	$0D

übernehmen, was wiederum die Adresse $030D00 ergibt, also Adresse 199936, die sich glatt durch 256 teilen läßt. Nun ist eine Lores - Zeile 160 Bytes lang und acht Zeilen 1280 Bytes, dies ist

das kleinste, gemeinsame Vielfache von 256 und 160, da 5*256 ebenso 1280 ergibt, somit ist auf jedem normalen ST ein Hardwaremäßiges acht Zeilen vertikales Scrolling möglich. Wenn man nun den Bildschirmspeicher sehr schnell scrollen möchte, ist eine Zeile pro VBL eh ein bißchen wenig und acht Zeilen in manchen Fällen gerade noch erträglich und ohne größeres Kopieren von Speicher zu realisieren.

7.2.2.2 Horizontales Scrolling

Auf einem normalen ST mit 8 MHZ CPU und ohne Blitter ist es auf normalem Wege (Bitverschieben) unmöglich, horizontales Finescrolling (bei jedem Bildaufbau eine Spalte) über den ganzen Bildschirm und vier Planes zu realisieren. Man schafft allerhöchstens 1/3 des Bildschirms. Das liegt an der, wie schon erwähnt, ineinander verschachtelten Struktur der Bitplanes im Bildschirmspeicher.

Scrolling durch Bitverschieben:

16 Farben

ersten 16 Pixel — zweiten 16 Pixel

1 Bit verschieben

1.Wort 1111111111111111 | 5.Wort 1111111111111111 etc.
2.Wort 1111111111111111 | 6.Wort 1111111111111111
3.Wort 1111111111111111 | 7.Wort 1111111111111111
4.Wort 1111111111111111 | 8.Wort 1111111111111111

Schemtische Darstellung des horizontalen Scrolling via Bitverschieben:

Beispielroutine dazu auf Diskette !

Aber wie gesagt, das ist keineswegs schnell genug für Spiele ! Für Spiele empfiehlt sich eine andere Methode, "da der Spieler sonst beim Spielen einschläft".

Wenn man den zu bewegenden Screen 16 mal verschoben im Speicher hält (16*32k (Bildschimspeicher ST - lores) = 256k), also alle Pixel schon mal irgendwo verschoben liegen, muß man zum größten Teil dann nur Wörter bzw. Langwörter in den Bildschirmspeicher kopieren und das schafft sogar ein ganz normaler 68000er 8MHz alle 20ms (50Hz). Die oben angestellte Rechnung mit einem benötigten Speicherbedarf von 256k zeigt schon den Nachteil dieser Methode an. Für den Falcon (640*480*256 Farben) würde dies z.B. bedeuten, über 4MB Zwischenspeicher zum Scrollen zu benutzen (wie gut, daß das Ding einen schnellen Blitter hat), "aber was tut man nicht alles für ein softes Scrolling". Viel Speicher kann man sich bei dieser Technik sparen, wenn der zu scrollende Hintergrund aus Mustern besteht, die z.B. nach dem vierten Verschieben genauso wie am Anfang aussehen oder wenn nur bestimmte Teile des Bildschirms scrollen, z.B. nur die untere Hälfte und im oberen Teil des Bildschirms ein blauer Himmel mit Sonne und Anzeige für Punkte und Spieler etc. stillsteht.

Mit der oben beschriebenen Technik kann man natürlich nicht nur scrollen, ein im Speicher 16 mal (um ein Pixel) verschobenes Sprite kann man nicht nur Zeile für Zeile aus einem der 16 Pufferspeicher kopieren sondern auch Zeile für Zeile aus unterschiedlichen Puffern, um somit Effekte wie "wellige" Sprites (bekannt aus einigen UNION - Demos) zu erzeugen.

7.2.3. Vergrößern von bitmapped Graphics

Bitmapped bedeutet pixelorientiert, das Gegenstück ist Vektorgrafik. Die Techniken, die ich hier kurz vorstellen möchte, kann man verwenden, um eine schnelle Lupe für ein Grafikprogramm zu entwerfen oder auch um Sprites wie bei Wing Commander zu vergrößern oder zu verkleinern.

7.2.3.1 Vertikales Vergrößern (in Y - Richtung)

Fangen wir mal mit dem Einfachsten an, das Verdoppeln stellt immer noch die leichteste Disziplin dar. Dazu kopiert man das zu verdoppelnde Objekt Zeile für Zeile in den Zielspeicher, wobei man jede Zeile im Zielspeicher doppelt kopiert.

Man kann allerdings die ganze Sache noch ein wenig flexibler gestalten, wenn man in einem Register einen Zähler einrichtet, der angibt, wievielmal die jeweilige Quell-Zeile ins Ziel kopiert wird, z.B. 2*,3*,4*.

Hier eine einfache Veränderung des Listings von 7.2.2.1; dieses Listing läßt jetzt schon Vergrößerungen u. andere Effekte zu. Es läßt sich auch zum Zusammenkopieren der Planes zweier Bildschirme benutzen (erste Zeile - erster Screen, zweite Zeile - zweiter Screen) und somit zum Aufbau des virtuellen Bildschirms zum "Hardwarescrollen" von STE/Falcon.

```
**** Kopieren von n Zeilen eines Lores - Screens *************

;erweitert durch Offset zur nächsten Zeile
;ermöglicht Vergrößerungseffekte u. Spiegelungen

* In A0 befindet sich die Anfangsadresse Quelle
* In A1 die Anfangsadresse Ziel
* D0 Zähler für die zu übertragenden Zeilen
* D1 Offset zur nächsten Zeile für Source
* D2 Offset zur nächsten Zeile für Destination


**************** Kopieren einer Zeile **************

zeile:

 movem.l (A0)+,D3-D7/A2-A6     ; die ersten 11 Longs lesen (Quelle)
 movem.l D3-D7/A2-A6,(A1)      ; schreiben (Ziel)
 lea    40(A1),A1              ; Addiert 44 Bytes zu A1 dazu
 movem.l (A0)+,D3-D7/A2-A6     ; nochmal 11 Longs lesen (Quelle)
 movem.l D3-D7/A2-A6,(A1)      ; schreiben (Ziel)
 lea    40(A1),A1
```

```
movem.l (A0)+,D3-D7/A2-A6   ; nochmal 11 longs lesen (Quelle)
movem.l D3-D7/A2-A6,(A1)    ; schreiben (Ziel)
lea    40(A1),A1
movem.l (a0)+,D3-d7/a2-a6   ; die Letzen 40 Bytes
movem.l d3-d7/a2-a6,(a1)
lea    40(A1),A1

***addition des Offsets (Abstand) zu nächsten Zeile *********

adda.l D1,A0        ;addiert Offset Source
adda.l D2,A1        ;addiert Offset Destin.

*************************************************************

dbra    D0,zeile

*************************************************************

END
```

Da aber einfache Vergrößerungen z.B *2,*3,*4,*5 für die meisten Zwecke nicht fein genug abgestuft sind, es in Maschinensprache auf einem 68k bis 68030 ohne FPU keine Floating Point Arithmetik gibt, muß man sich mit einem kleinen Trick helfen, um Vergrößerungen wie z.B 2.5 oder 3.12 zu erzeugen.

Zuerst zur Theorie; benutzt man nicht ganzzahlige Faktoren bei der Vergrößerung von Bildern wird nicht Zeile für Zeile gleichmäßig vergrößert (wiederholt kopiert) sondern jede Zeile unterschiedlich häufig von der Quelle zum Ziel kopiert.

z.B.: Faktor 2.5

Zeile 1: 2 mal
Zeile 2: 3 mal

bei noch "krummeren" Faktoren z.B. 2.125 kann es sein, daß man kaum noch ein Muster bei den Vergrößerungen pro Zeile erkennt.

Hier ein Routine, die dieses Problem löst.

```
******** Kopieren von n Zeilen eines Lores - Screens *************

* In A0 befindet sich die Anfangsadresse Quelle
* In A1 die Anfangsadresse Ziel
* In A2 Endadresse des Ziels (für clipping)
* D0 Zähler für die zu übertragenden Zeilen
* D1 > 65535 = Vergrößerung, D1 < 65535 Verkleinerung

**************** Kopieren einer (Quell) Zeile **************

zeile:
    swap    D1                  ; alles was >65535 ins
                                ; niederwertige Wort
wie_oft:
    cmp.w   #0,D1               ; keine ganze Zeile ?
    beq.s   aus                 ; dann raus
    cmpa.l  A2,A1               ; vergleiche jetzige mit Endadresse
    bge.s   out                 ; >= dann raus (sonst 2 Bomben)
    movem.l 0(A0),D3-D7/A3-A6           ; die ersten 9 Longs lesen (Quelle)
    movem.l D3-D7/A3-A6,0(A1)           ; schreiben (Ziel)
    movem.l 36(A0),D3-D7/A3-A6          ; nochmal 9 Longs der Zeile
    movem.l D3-D7/A3-A6,36(A1)          ; schreiben (Ziel)
    movem.l 72(A0),D3-D7/A3-A6          ; nochmal 9 Longs der Zeile
    movem.l D3-D7/A3-A6,72(A1)          ; schreiben (Ziel)
    movem.l 108(A0),D3-D7/A3-A6         ; nochmal 9 Longs
    movem.l D3-D7/A3-A6,108(A1)         ; schreiben Ziel
    movem.l 144(A0),D3-D6               ; die letzten 4 Longs
    movem.l D3-D6,144(A1)               ; schreiben Ziel
    lea     160(A1),A1                  ; nächste Zeile des Ziels
    dbeq    D1,wie_oft                  ; solange Zeilen
*****************************************************************
aus:
    sub.w   D1,D1               ; D1 ist dann -1 also löschen
    swap    D1                  ; niederwertiges wieder nach oben
    add.l   D2,D1               ; Faktor dazuzählen
    lea     160(A0),A0          ; nächste Zeile Quelle
    dbra    D0,zeile            ; solange Zeilen>=0
out:
*********************************************************

    END
```

Der Unterschied zwischen dieser und der Kopierroutine vo 7.2.2.1 ist gar nicht so groß. Zum einen erkennt man natürlich, daß die Adresse für Ziel in 'A1' erst nach dem Kopieren der kompletten Zeile auf die Anfangsadresse der nächsten Zeile gesetzt wird zum anderen, daß die Quelladress erst nach n - maligem Kopieren (wobei n im höherwertigen Wort von 'D1' steht) auf die nächste Zeile gesetzt. Wenn das höherwertige Wort 'D1' Null enthält, wird gar keine Zeile kopiert.

Warum im höherwertigen Wort ?

HW=Höherwertig
NW=Niederwertig
in D2 steht z.B. $1A000

		HW	NW
D1	$	0000	0000
D1=D1+	$	1	A000

Tausche HW mit NW (in Assembler **SWAP D1**)

dann steht in D1

$ A000 0001

da ich beim Vergleichen 'cmp.w <-' nur das Register oder die Konstante mit dem niederwertigem Wort von 'D1' vergleiche und der Befehl 'dbcc' auch nur das niederwertige Wort von 'D1' herunterzählt, beibt dessen höherwertiger Teil unangetastet. Ich "swappe" dann wieder die Worte von 'D1' und addiere wiederum die in 'D2' stehende Konstante zu 'D1'

```
clr.w    D1 ---- >   D1 = $A000  0000
swap     D1 ---- >   D1 = $0000  A000
add.l    D2,D1- >    D1 = $0002  4000
```

d.h die nächte Zeile wird nicht einmal sondern zweimal kopiert.

Zum Thema Sounderzeugung (B.4.2.2) erkläre ich diese Technik nochmals auf eine andere Art, vielleicht wird es dann etwas klarer.

7.3.2.2 Vergrößern in X - Richtung

Wie man im letzten Abschnitt gesehen hat, ist das Vergößern in Y -Richtung recht leicht, das es in X-richtung nicht ganz so einfach ist, liegt wieder an der Anordnung der Bitplanes. Trotzdem ist es möglich, mit relativ geringem Aufwand mit annehmbarer Geschwindigkeit zu vergrößern. Der Trick ist dabei eine Tabelle, die aus einem Byte ein Wort macht und somit eine Verdopplung der Pixel in X - Richtung verursacht.

Die ersten Elemente einer solchen Tabelle können dann in etwa so aussehen:

2.Element	3.Element	256 Element
....,%0000000000000011,	%0000000000001111,.......,	%1111111111111111

Ist also das erste Byte einer Bitplane eine %0000001, multiplziert man den Wert mit 2 (um ein Bit nach links schieben) und holt dann das entsprechende Wort aus der Tabelle in dem Fall also ein Wort, das so %0000000000000011 aussieht. Das macht man dann für alle Bitplanes des entsprechenden Bereichs. Wenn man dann Vergrößerungen wie mal 4 oder mal 8 vornehmen möchte, kann man dann die entsprechende Prozedure des Heraussuchens aus der Tabelle mehrmals durchführen, um allerdings Sprites oder ähnl. um krummere Werte als mal 4; mal 8 wie z.B mal 3 oder gar mal 2.5 zu vergrößern, müßte man mehrere Tabellen und ein etwas komplexeres Verfahren verwenden.

7.4 Diverse kleinere Effekte

7.4.1 Spiegelungen der Bitplanes an der X - Achse

Bestes Beispiel für so einen Effekt ist das Spiel 'NEBULUS' oder das Anfangsbild von 'Turrican 2'; aber auch in Demos kann man oft gespiegelte Scroller (scrollende Schrift) und ähnliches sehen. Im Prinzip ist das kein Problem, man kopiert Zeile für Zeile aber zählt die Zieladresse nicht nach oben sondern fängt am Ende des Zielspeichers an und erniedrigt die Zeiladresse der nächsten Zeile statt sie zu erhöhen. Natürlich liegen Quelle und Ziel meist beide im sichtbaren Bereich (Quelle z.B. im oberen und Ziel im unteren). So kann man z.B. im oberen Bereich des Bildschirms viele Sprites etc. anzeigen und dann die genze Sache komplett nach unten kopieren und dabei spiegeln, was denn wesentlich weniger Rechenzeit als ein Neuaufbau benötigt (selbst wenn man die Sprites alle schon gedreht im Speicher hätte).

7.4.2 Schatten

Wenn man z.B. in LORES (320*200, 16 Farben) die ersten 8 Farbregister ausgewählt hat und dann die nächsten acht Register mit der Farben in der halben Helligkeit gewählt hat, kann man durch 'OR' Verknüpfung mit der Farbe 8 bzw. VDI Farbe 9 einen Art Schatten erzeugen.

Beispiel:

1.Plane	2.Plane	3.Plane	4.Plane
%1001110010101010	%1011001111000011	%1111111100000000	%0000000000000000
	OR		
%0000000000000000	%0000000000000000	%0000000000000000	%1111111111111111
%1001110010101010	%1011001111000011	%1111111100000000	%1111111111111111

Dieses Beispiel zeigt, wie so ein Effeckt bei der Verknüpfung der ersten 16 Pixel mit der entsprechenden Bitkombination von Farbe 8 bzw. VDI Farbe 9.

7.4.3 Sternenfelder

Dieser Effekt aller Star Raiders ist relativ leicht zu erzeugen. Man wählt per Zufall die Flugrichtung und Geschwindigkeit für einen Stern aus und läßt ihn vom Mittelpunkt zum Rand des Bildschirms bewegen. Wenn man das mit genügend vielen Sternen macht, ist der Eindruck perfekt. Zusätzlich kann man auch Sterne mit größerer Geschwindigkeit von vornherein weiter nach außen plazieren, was die Wirkung noch etwas verbessert. Listing dazu befindet sich auf der Diskette zu Buch in 'KAPA7.TOS'.

A.8. Wie bleibe ich kompatibel

(zu allen jetzigen und kommenden Ataris)

Vorwort

Wer sich einen TT oder Falcon030 gekauft hat, wird feststellen, daß man nicht alles auf Anhieb zum Laufen bringen kann. Gerade bei Spielen und Demos scheint es Brauch zu sein, den Geräten das Programm auf den Leib zu schneidern und das so eng, daß es auf einem geringfügig modifizierten Gerät (TOS an anderer Position oder 16 MHz Speeder) einfach nicht mehr "zum Laufen zu bringen" ist. Ein Programmierer, der darauf stolz ist, daß er ein Programm trotz einer "zu schwachen" Hardware "zum Laufen" gebracht hat, darf erst dann auf sich stolz sein, wenn er das Programm so geschrieben hat, daß es auf jeder möglichen Konfiguration und nach den Richtlinien des Herstellers (auch wenn's manchmal schwer fällt) programmiert hat. Man kann sich nicht damit herausreden (wie die Programmierer auf dem Amiga das bis vor kurzem noch mit Recht konnten), daß es keine Richtlinien gäbe und keine Zeitung über diese berichtet hätte. Es sind mehr als ein Dutzend Berichte über saubere Programmierung unter TOS in den unterschiedlichsten Fachzeitschriften erschienen. Also haltet euch daran!

8.1. Die Hardware

8.1.1 Schreibe nie hinter den Bildschirmspeicher

(denn der Atari könnte drei Bomben "werfen")

Die normale MMU des Atari ST(E) kann nicht mehr als 4 MB RAM adressieren, große Teile hinter diesem normalerweise (es geht mit etwas Hardware doch) nicht zu adressierenden Speichers sind durch die MMU gesperrt ! Das bedeutet wiederum, daß bei einem Versuch diesen Speicherbereich zu lesen oder zu beschreiben, die CPU von der MMU (Memory Management Unit - "Speicherverwal-

ter") einen Addressfehler gemeldet bekommt, weil sie diese Addresse ja gar "nicht kennt", was wiederum einen Programm - Absturz zur Folge hat.

Lösung:

1) Nie über den Bildschirmspeicher hinweg lesen oder schreiben !

2) Falls man ein solches Programm laufen lassen möchte, ist die einfachste Möglichkeit (aber nicht sehr sauber), den Bildschirmspeicher um einige hundert Bytes nach unten zu verschieben. Diese ist aber sehr problematisch, da das Betriebssystem den Speicher, wo dann der Bildschirmspeicher sitzt, vielleicht schon benutzt. Besser man schreibt sich ein Programm, das den Memtop verschiebt, siehe *'KAPA8.TOS'* - *MEMORY.S*.

Dieses Problem tritt (nur!) bei ST(E)'s auf, da die MMU des TT's auch auf Speicher hinter der 4MB-ST-RAM Grenze zugreifen kann, man kann sich mit Hilfe der programmierbaren MMU des 68030 im Prinzip an fast jede beliebige Stelle Speicher legen! Jedoch natürlich nur, falls man genügend hat, denn Speicher ist wie ein Bettuch, wenn man an einer Seite zieht, fehlt es an einer anderen.

8.1.2 Das Syncmode Register des TT

(oder wie der Bildschirm plötzlich ganz schwarz wurde)

Leider hat Atari bei der Entwicklung des TT - Shifters wohl einen "klitzekleinen" Design Fehler mit großer Wirkung gemacht. Das Bit 0 im Syncmode Register des TT, das auf externe Synchronisation (für Genlocks etc.) umschaltet, reagiert genau anders herum wie bei ST (siehe Teil 2, Kapitel 1), damit wiederum haben die Programmierer nicht gerechnet, die in ihrem Programm auf die augenfreundlicheren 60 Hz Bildwiederholfrequenz (wenn im ST das Bit 1 des Syncmode Registers gelöscht wird) umschalten und dabei das Bit 0 des Syncmode Registers auch gleich löschen, denn dann

wartet der TT auf externe Synchronisation und darauf kann er dann meist lange warten. Dieses kommt einem Absturz des Rechners gleich, da kein VBL mehr ausgeführt wird (fast nichts geht mehr).

Lösung:

1) 60Hz Umschaltung mit ANDI.B #2,$ffff820A.w durchführen !!!

2) Um Programme starten zu können, die eine solche Umschaltung beinhalten, das Bit 0 des Syncmode Registers per TT - Timer immer wieder setzen, z.B. dem Programm aus *KAP8.TOS, SYNCFIX.S.*

8.1.3 Probleme mit Syncscrollern ???

Wer sogenannte Syncscroller in eigenen Programmen benutzt, ein Verfahren um mit Hilfe des Shift Mode und Sync Mode Registers durch geschicktes Umschalten bei ST und damit Öffnen der Bildschirmränder bestimmter Zeilen, das nicht vorhandene Video Low Byte simuliert (siehe Kapitel über Syncscrolling), also wer Techniken wie Öffnen der Bildschirmränder und Syncscrolling wirklich nötig hat, sollte Routinen anbieten, die alternativ bei unbekannter Hardware oder möglichen Timing Schwierigkeiten, z.B. durch 16Mhz Speeder, die gleiche Funktionen erfüllen, auch wenn das Programm in einem solchen Fall vielleicht nicht mehr ganz so toll aussieht.

8.1.3 Man muß auch warten können

(Probleme mit Warteschleifen)

Viele Programmierer, die qualitativ hochwertige Spiele von Systemen mit mehr Farben (z.B. PC mit VGA) oder einfacher zu erzeugendem Scrolling (z.B. AMIGA) auf einen ganz normalen ST umsetzen "müssen", realisieren dies durch exaktes Synchronisieren des Prozessors mit dem Bildschirmaufbau. Diese Programme laufen meist nur so lange der Prozessor mit 8 MHz getaktet ist, bei

jeder Änderung der Taktfrequenz ändert sich möglicherweise das Timing des Prozessors total. Das bedeutet wiederum, daß irgendwelche Bitschiebebefehle z.B. bei 16 MHz Speedern möglicherweise doppelt so schnell ausgeführt werden, aber NOP's (no operation - ein Befehl, der nichts als wartet), nicht schneller als zuvor abgearbeitet wird. Beim nächst höheren Prozessor z.B. 68030/040 können sich die Verhältnisse der Prozessor-Befehle zueinander wiederum stark verändert haben.

Lösung:

1) Benutze die Timer, um z.B. Farbumschaltungen an der richtigen Stelle einsetzen zu lassen oder Warteschleifen zu ersetzen, siehe auch dazu mein Kapitel über Timer Programmierung.

2) Starte mein '*VERZTT.PRG*' in *KAPA8.TOS*, das mittels des TT Timers A des MFP den Prozessor so oft unterbricht, daß er nur ca. 1/4 bis 1/6 seine Leistung bringt und somit das Spielen bestimmter Programme auf dem TT erst wieder möglich macht. Für den Falcon gibt es da eine elegantere Lösung, denn dieser läßt sich auf ca. 1/2 seiner Leistung drosseln, wenn man das Bit 0 im undokumentierten Register $FFFF8007 löscht. Das Verändern der Speicherstellen $FFFF8006/7 hat eine große Wirkung auf das Verhalten des F030, da diese Register jedoch undokumentiert sind und ich deren Wirkung noch nicht komplett entschlüsseln konnte, hilft nur probieren. Trotz alledem befindet sich ein Patch ! '*VERZFAL.PRG*' Programm (und diese dürfen bekanntlich fast alles benutzen) auf Diskette, das den Falcon030 seiner Geschwindigkeit beraubt.

8.1.4 Andere Prozessoren - anderes Stackformat

8.1.4.1 Stackformat - Was ist das

Je nach Interrupt legen die 680X0 Prozessoren unerschiedliche Daten auf den Stack, um später den regulären Ablauf des Programms wieder aufnehmen zu können. So hat der Stack, je nach dem welcher Interrupt auszuführen ist, ein anderes Format.

8.1.4.2 Das Stackformat

Motorola redet bei seinem Prozessoren der 68000er Familie von Usermode Kompatibilität. Einen einzigen "Design Fehler" (die Ausnahme von der Regel) hat sich Motorola dennoch geleistet. Der Befehl MOVE From SR (MOVE SR,<ea>), der das Statusregister in den Speicher kopiert, ist für alle Prozessoren, außer dem 68000, ein Privilegierter Befehl (kann nur im Supervisor Mode ausgeführt werden).

Im Supervisor Mode sieht es mit der Kompatibilitiät zwischen den 680X0 Prozessoren nicht ganz so "rosig" aus, allerdings liegen die Unterschiede an einer Stelle, an der ein "normaler" Programmierer eigentlich nicht zu "rütteln" hat, und zwar das Stackformat bei Interrupts. Motorola ändert zwar bei der Entwicklung eines neuen Prozessors sein Stackformat nicht immer, aber immer öfter !

Nun, gehen wir etwas ins Detail, am wichtigsten für uns Programmierer ist wohl das Default (übliche) Stack Format. Dieses tritt am allermeisten auf.

Default Stack Frame !

hat bei 68010/20/30/40er Format Code %0000

Byte	68000	68010/20/30/40
0	Status Register	Status Register
2	PC oberes Wort	PC oberes Wort
4	PC unteres Wort	PC unteres Wort
6		Bit 15-12 Format Code %0000
		Bit 11- 0 Vector Offset

- Format Code (oberste 4 Bit des 4. Worts) gibt Stack Format des jeweiligen Interrupts an

Code	Länge des Stackformats in Bytes	tritt z.B. auf bei
%0000	8	Default
%1000	58	Bus Fehler 68010
%1010	32	"" "" "" 68020/30 Short
%1011	92	"" "" "" "" "" Long 68010/20/30
%0010	12	Division by Zero u.v.m 68020/30/40
%1001	20	Line F 68020/30
%0001	8	Exception Stack Frame
%0111	60	Access Error Stack Frame 68040
%0011	12	Floating-Point Post-Instruction 68040

- Vector Offset (unterste 12 Bit des 4. Worts) gibt den Abstand des aufgetretenen Interrupts in Bytes zur Vector Basis Adresse (steht im Vector Basis Register) an. Man kann mit Hilfe dieses Offsets z.B. einen Trap Dispatcher schreiben, der alle ankommenden Exceptions an die entsprechenden Betriebssystem-Routinen weiterleiten kann, da man aus dem Offset/4 ja die Nummer des Traps errechnen kann.

Die neueren TOS Versionen erkennen das geänderte Stack Format und passen sich an (ab TOS 1.06). So läßt sich zum Beispiel schon in einen 1040 STE ein 68010 einbauen (der bei gleichem Takt schon ein wenig schneller ist) allerdings dürften dann einige Debugger und Demos nicht mehr laufen.

Ab TOS 3.0 läßt sich das Stack Format über die System Variable _longframe, die an der Adresse $59e liegt,abfragen - wenn der Inhalt der Variable > 0 ist, kann man davon ausgehen, daß man sich in einem System mit Prozessor > 68000 befindet. Diese Variable dürfte sogar von den meisten modifizierten TOS-Versionen der Hersteller Beschleunigerkarten entsprechend gesetzt oder gelöscht werden, da ein großer Teil von Ataris Systemsoftware nur bei entsprechendem Eintrag in _longframe funktioniert.

8.1.5 Die Caches

- Cache, was ist das ? *(kurze Definition)*

Cache ist ein schneller Zwischenspeicher, auf den der Prozessor schneller als auf das restliche Ram zugreifen kann. Beim Zugriff auf den Speicher testet die Logik auf dem Cache, ob sich der entsprechende Speicherinhalt nicht auf dem Cache befindet, z.B. wenn man kurz zuvor schon einmal auf dieselbe Adresse zugegriffen wurde, und greift, falls das zutrifft, auf den schnelleren Cache-Speicher zu.

- Probleme mit DMA Bausteinen bei eingeschaltetem Cache

Sowohl der 68030 im TT und Falcon030 als auch der MEGA STE besitzen Cache - Speicher. Beim 68030 ist es ein interner Prozessor Cache, der aus je 256 Byte Daten- und Befehlscache besteht. Im MEGA STE dagegen befindet sich ein externer Cache der als Puffer zwischen Prozessor und Speicher um die Performance einer 16 Mhz getakteten CPU auf das auf 8 Mhz getaktete Board des STE zu bringen. Ein 16 Mhz getaktetes Board hätte Atari wohl im Vergleich zum Nutzen zuviel gekostet. Außerdem versagen auf dem Mega STE auch die meisten der vielen, relativ unsauber geschrieben Spiele nicht ihren Dienst. Wenn mal eins nicht funktionieren sollte, hilft oft TOS 1.0 oder 1.04 ins RAM zu laden. Aber nun zurück zum Cache, bei direkter Programmierung der Hardware kann es bei angeschaltetem Cache zu Problemen kommen, die so ohne weiteres nicht ersichtlich sind. Ein Beispiel wäre die direkte Programmierung des Floppycontrollers oder der ~~SCSI~~/ASCI DMA (direkt memory access / direkter Speicher Zugriff). Man programmiert also den DMA Chip, dazu einige Bytes von den Sektoren einer Diskette zu lesen, dieser legt dann diese Bytes selbständig (daher auch DMA) in den Speicher. Währenddessen hat der Prozessor fast keinen Zugriff auf den Speicher. Der Prozessor bekommt erst dann den Bus (also den Zugriff auf den Speicher) wieder, wenn der DMA-Chip eine Pause macht oder die Übertragung in den Speicher beendet wurde. Nun steht im Cache des Prozes-

sors möglicherweise noch der vorherige Inhalt des Speichers, den man zuvor per DMA verändert hat, das wiederum kann zu fehlerhaft ausgeführten Programmen führen (sprich Bomben), da ein Teil des Programms aus dem Cache stammt, ein anderer Teil von der Diskette. Diese Probleme treten immer dann auf, wenn Daten im Speicher ohne Wissen des Prozessors geändert werden z.B. durch Blitter oder anderen DMA-Bausteinen. Bei den heute gängigen Beschleuniger-Karten wird zumindest der externe Cache gelöscht, wenn der Prozessor den Bus abgibt (das passiert z.B. bei Benutzung des DMA Controllers oder des Blitters). Es ist jedoch schwer möglich, den 68020/30 in seinem Cachehandling so zu beeinflussen. Beim 68040 wurde daher sogenanntes "bus snooping" implementiert, der "intelligente Cache" des 68040 "horcht" am bus und falls ein anderer Prozessor seinen Schreibzugriff bekannt gibt, löscht er automatisch die entsprechenden Cache-Einträge.

Bei Benutzung eines 68020/30 sollte man daher entweder den Cache vor dem DMA Betrieb ausschalten bzw. für ungültig erklären oder Überschneidungen von Cache und DMA durch intelligente Programmierung vermeiden, besser noch (weil das die sauberste Lösung ist) nur Betriebssystem-Funktionen verwenden und die Hardware nicht direkt programmieren (wenn das immer so leicht ginge !).

- Selbstmodifizierender Code

Zuerst möchte ich mal erklären, was das ist !

Alle Atari Rechner sind sogenannte "von Neumann" Rechner (die MAC's, Amigas, PC's, NEXT Rechner auch). Bei dieser Art von Rechnern liegen Programmcode und Daten im selben Speicher. Nun, der Prozessor kann Daten verändern, Speicherzellen löschen in denen ein Datum (Einzahl von Daten) vorhanden ist und das an jeder Stelle des Speichers! Also auch an den Stellen, wo sich ausführbarer Programmcode befindet. Somit ist es also möglich, wärend des Ablaufs eines Programms das Programm zu ändern um somit z.B. die Bedingungen oder Sprungoffset's von Branch-Befeh-

len zu ändern und somit vielleicht Stellen im Programm zu modifizieren, die mehrmals ausgeführt werden müssen. Mit Selbstmodifizierende Programmen ist es zwar sehr oft möglich, aus dem Rechner "das Letzte rauszuholen", aber solche Programme sind zum einen sehr unübersichtlich und damit fehlerträchtig und zum anderen Vertragen sie sich nicht mit Caches und Pipelinig (Technik mehrere Befehle im Prozessor, wie am Fliesband, zu bearbeiten). Nun hat man wieder zwei Möglichkeiten, Programme zu schreiben, die auf allen Atari Rechnern (ab ST aufwärts) laufen! Richtungsweisend wäre es, Algorithmen (Rechenvorschriften) zu schreiben, die schnell genug sind und doch auch ohne die Modifizierung des eigenen Programmtextes auskommen, aber das ist gelegentlich 'fast' unmöglich. Andererseits könnte man zwei Routinen schreiben, die man je nach Prozessortyp verwenden könnte.

8.2. Das Betriebssystem

8.2.1 Generelles

ATARI ist es zumindest gelungen, jede neue Betriebssystem Version abwärts kompatibel zu halten, das bedeutet jedoch, daß mögliche neue Calls auf den kleineren Versionsnummern nicht laufen.

Ein Beispiel ist die erweiterte Fileselect Box ab TOS 1.4 (AES 91), jede Verwendung in einer älteren TOS-Version erzeugt Bomben! Man sollte bei Verwendung solcher neuen Funktion des Betriebssystems zuvor die AES Version Nummer, die bekommt man beim 'appl_init()' (AES 10) im 'Global' Feld zurückgeliefert. Nur die TOS-Versionsnummer abzufragen, würde in diesem Fall leider nicht ausreichen, da der Benutzer ja ein externes AES ins RAM geladen haben könnte.

Auch hat ATARI die Lage des Betriebssystem von $FC0000 auf $E00000 verlagert, womit scheinbar einige Spiele und Demoprogrammierer niemals gerechnet haben! Obwohl schon durch die interne Struktur des Betriebssystem (außerdem lag es ja bis Mitte

1986 im RAM) jedem klar sein mußte, das Positionsänderungen ATARI wirklich keine Probleme bereiten dürfte. Die Geräte, die unter TOS laufen, sind eben keine "Homecomputer", denn bei diesen Rechnern und nur bei diesen, ist das Springen über absolute Adressen nötig. Auch sind undokumentierte Systemvariablen nicht ohne Grund undokumentiert! Im Klartext heißt das, immer nur das benutzen, was Atari auch wirklich dokumentiert hat, Ausnahmen sind dabei natürlich div. Patchprogramme und systemnahe Software; diese dürfen natürlich "fast alles" benutzen, da sie nur mit einer speziellen Rechnerconfiguration laufen müssen.

8.2.2 Line A

Für den Prozessor sind alle Befehlswörter, die mit einem $A beginnen unbekannt, er löst eine Ausnahmebehandlung (Exception) aus.

Das Line A Interface ist die softwaremäßige Emulation eines imaginären Grafik-Chips. Über diese Emulation erledigt das VDI seine Grafikausgaben. Nun unterstützen die wenigsten VDI Treiber für z.B. Grafikkarten die Lina A Grafik, das bedeutet wiederum, daß Programme, die Line A Grafiken benutzen, nicht auf diesen Karten laufen, was gegen die Verwendung von Line A Variablen spricht.

In den Dokumentationen von Atari zum TT wird eindeutig empfohlen, bei Grafik Ausgaben das VDI zu nutzen, da Line A nur zur Kompatibilität mit älteren Systemen ins TT-TOS implementiert (eingebunden) wurde. Im Klartext, Hände weg von LINE A, denn seit es div. Programme gibt, die das VDI stark beschleunigen und fast 100% kompatibel zum Original VDI sind, ergibt es keinen Sinn, in Standart Aplicationen (auch in Spielen) die LINE A Routinen zu benutzen.
Außerdem sind die LINE A Routinen nicht reintrant, d.h. es kann zu Problemen kommen, wenn zwei Programme (z.B. unter Multitos) auf die Line A Variablen zugreifen, da bei den LINE A Routinen nur ein Speicherbereich zur Parameterübergabe vorhanden ist, so daß sich die zwei Applikationen, falls mitten beim Aufruf eine Prozesswechsel stattfindet, gegenseitig die Parameter zerstören.

8.2.3 LINE F

Die Befehle der FPU (Floating Point Unit bzw. mathematischer Coprozessor) beginnen alle mit einem $F, wenn kein Coprozessor vorhanden ist, wird ein Interrupt ausgeführt. In diesem Interrupt ist es dann möglich, die fehlenden Coprozessor-Befehle zu emulieren (nachzuahmen).

Obwohl der Sinn und Zweck der LINE F von Motorola von Anfang an bekannt gegeben wurde, hat selbst ATARI zu Beginn die LINE F für AES Aufrufe gebraucht (bis incl. TOS 1.04). Da ist es nicht verwunderlich, daß auch einige Programmierer auf die Idee gekommen sind, den LINE F Vektor auf einen eigenen Dispatcher (Verteiler) zu richten und von diesem Punkt aus eigene Routinen aufzurufen. Nun spätestens mit dem Erscheinen des TT's mit eingebauten Coprozessors laufen diese Programme nicht mehr, da statt der Dispatcher der Coprozessor aufgerufen wird, der dann natürlich die vorgesehene Funktion nicht erledigt.
"Leider" läst sich der Coprozessor nicht so einfach abschalten, so daß die Programme die LINE F für eigene Zwecke nutzen auf dem TT ebenfalls ihren Dienst verweigern.

Jedes neue Programm sollte es vermeiden LINE F für andere Zwecke als Coprozessor Emulationen zu nutzen !

Jede Zuwiderhandlung wird mit lebenslanger Inkompatiblität bestraft !!!

8.2.4 Systeminformationen mit Hilfe des Cookie Jar

("Keksdose", oder wie Dir Kekse helfen können)

Ab TOS 1.06 wurde in das TOS das sogenannte "Cookie Jar" installiert. Der Zeiger auf die Cookies befindet sich an der Adresse $5a0. Ab TOS 2.06 bzw. 3.06 konfigurieren sich die Cookies von selbst, d.h. das TOS testet die im Computer befindliche Hardware und richtet entsprechend die Cookies ein. Aber auch Programme, die nach ihrem Start resident (nach "Beendigung" des Programms sind sie immer noch im Speicher) im Speicher bleiben, z.B. Ram-

disk und Mausbeschleuniger, können sich in das Cookie Jar eintragen, um einem anderen Programm zu zeigen, daß sie präsent sind und entsprechend konfiguriert werden können.

Ein gutes Beispiel hierfür ist z.B. NVDI, dessen Accessory, das man zum Einstellen der Parameter benötigt, sich nur installiert, wenn NVDI auch installiert ist und das erkennt es natürlich an dem Eintrag im Cookie Jar.

Funktionsweise

Falls sich an der Adresse, auf die $5a0 zeigt, eine Null befindet, ist noch kein Cookie Jar installiert. Das testende Programm läuft also in einem Atari ab, dessen BIOS keine Cookies installiert hat oder irgend jemand hat gemeinerweise alle Cookies gelöscht...

Ein Cookie (Eintrag im Cookie Jar) besteht aus zwei Langworten. Das erste Langwort sollte eine eindeutige Nummer sein, die in irgendeiner Weise kenntlich machen soll, von welchem Programm das Cookie stammt z.B. 'NVDI'. Das erste Zeichen eines solchen aus ASCII-Zeichen bestehenden Strings (keine nationalen Sonderzeichen !) darf kein '_' (underscore, $5f) sein, da so das BIOS seine Cookies kennzeichnet. Der String darf keine Variante des Wortes 'Cookies' sein, da das laut Atari zu offensichtlich wäre und auf keinem Fall den Eintrag an sich beschreiben würde, es wäre eben so als würde man eine Variable in einem Programm 'var' nennen. Wenn man ein bestimmtes Cookie sucht, ist die Suche an der Stelle zu Ende, wenn das erste Langword Null ($00000000) ist, denn das ist dann das Ende der Cookie Jar.

Das zweite Wort ist dann die Information zum Cookie !

z.B. 1. Langwort 2. Langwort

BIOS: Standard Cookie Installation ab TOS 1.06

_CPU	0,10,20,30,40 (Dezimal) bedeutet 68010/20/30/40 etc.
_VDO	Video Hardware oberstes Wort 0 - ST, 1 - STE, 2 - TT, 3 - Falcon
_SND	Tonerzeuger BIT 0 - GI/Yamaha BIT 1 - stereo DMA BIT 2 - 16 Bit CODEC BIT 3 - DSP BIT 4 - Connection Matrix

q

Leider hat ATARI kein Bit für LMC 1992 und Microwire Interface im Sound Coockie definiert !

_MCH	Computertyp oberstes Wort 0 - 520 ST, 1040 ST 1 - STE 2 - TT 4 - Falcon

Die Abfrage der Cookies wurde in verschiedenen Beispiel Applicationen schon gezeigt (z.B. Argon 4) !
Das Cookie Jar ist also eine Möglichkeit um herauszufinden, auf welcher Plattform das Programm gerade läuft. Es gibt noch div. andere Cookies, die aber für Spieleprogrammierer in der Regel uninteressant sind.

8.3. MULTI TOS

das offizielle Multitasking Betriebssystem von Atari

Jeder, der jetzt nicht gerade Action Spiele schreibt oder schnelle Animation in seinem Programm benötigt, sollte versuchen, seine Programme in Fenstern laufen zu lassen. Gerade Adventures und Simulationen bieten sich an, so programmiert zu werden, daß ihre Grafikausgaben in Fenster erfolgen. Dazu gehört natürlich, nicht direkt in den Bildschirmspeicher zu schreiben, sondern daß Betriebssystem alle Ein/Ausgaben tätigen zu lassen. Das erhöht stark den Nutzen eines Spieles, da selbst die etwas "älteren Semester" gern mal ein "Spielchen" wagen, wenn der Rechner im Hintergrund druckt oder seinem User (Benutzer) andere wichtige Tätigkeiten abnimmt.

Nun gehört zur GEM Programmierung ein bestimmtes Grundwissen über AES und VDI, dieses zu vermitteln, würde die Seitenzahl dieses Buch nahezu verdoppeln, da es aber seit MULTI TOS sinvoll ist, mehr über das GEM zu wissen, wäre es empfehlenswert sich ein solches Buch (siehe Anhang B) zuzulegen und das auch für Spieleprogrammierer !

8.3.1 Benutze nie Speicher, der Dir nicht gehört

Unter Multi TOS ist es möglich, Speicherplatz, den man sich vom GEMDOS anfordert oder in dem das Programm läuft, bestimmte Attribute zu geben. Es ist damit möglich, laufende Prozesse voreinander zu schützen, aber auch unter bestimmten Bedingungen zugänglich zu machen. Das funktioniert effektiv allerdings nur bei Rechnern, die einen Prozessor mit/und PMMU (programmierbarer Speicherverwaltungsbaustein) beinhalten, wie TT und Falcon. Multi TOS kennt vier Arten von Speichern.

1. Privaten. Das ist Speicher, der nur von dem Programm (und dem Betriebssystem) benutzt werden kann.

2. Globalen. Dieser Speicher ist ungeschützt, jeder darf von ihm lesen und ihn beschreiben.

3. Super. Darf von anderen nur im Supervisor Modus angesprochen werden.

4. Privater (nur lese).
Jeder darf seinen Inhalt lesen, aber keiner darf ihn beschreiben.

Programme, deren Programmkopf nicht entsprechend modifiziert wurde, laufen "nur" in ihrem eigenen Speicher. Aber selbst wenn man die entsprechenden Bits im Programmkopf ändert, kann man damit allerhöchstens den eigenen Speicher für Programme zugänglich machen, andere Speicherbereiche können trotz allem geschützt sein ! Ein Zugriff auf fremden Speicher könnte somit zum "Russischen Roulette" werden. Also immer schön den Speicher, den man benutzen möchte, beim Betriebssystem verlangen. Nähere Informationen über Multi TOS sind im Anhang X zu finden.

8.4 Ataris Game/Entertainment Software Guidelines

Zur Veröffentlichung des Falcon030 hat Atari an die Entwickler von Spielesoftware erstmals Richtlinien zur Programmierung herausgegeben.

Um es dem Leser zu vereinfachen, habe ich diese ins Deutsche übersetzt.

Die folgenden Punkte sollen befolgt werden...

- Installierbar auf Festplatte

- Soll von jeder Bildschirmauflösung zu starten sein

- Der Benutzer sollte mit einer einzigen ausführbaren Datei konfrontiert werden, die dazugehörigen Daten, Highscore etc. sollen sich in einem dazugehörigen Ordner befinden.

- Dem Benutzer sollte es möglich sein, das Programm zu verlassen und den Desktop wieder so vorzufinden wie er ihn verlassen hat.

- Verwende den verbesserten Joystick für alle Spiele, für die Joysticks nötig sind; Regler wie der CX 40 sollten nicht unterstützt werden.

- Idealerweise, wo möglich, lassen Sie Ihr Spiel in einem Fenster laufen; passend für die Benutzer, die unter einer Multi - Tasking Oberfläche spielen wollen (und z.B währenddessen eine Datei von einer Mailbox übertragen)

- Wir nehmen an, daß die meisten Benutzer die 640*480 Punkte Auflösung mit 256 Farben nutzen, bitte denken sie daran.

- Wenn sie den VDI Aufruf, vr_trn_fm() (transform form) benutzen, können sie leicht Bild-Daten vom Standard-Format in das Format der gerade laufenden Auflösung wandeln.

Ich habe die Übersetzung so genau wie möglich und so frei wie nötig vorgenommen, um Mißverständnisse zu vermeiden.

8.4.1 Was hat das nun zu bedeuten

zu Punkt 1

Kein Diskettenkopierschutz aber entsprechende Installationsprogramme auf den Disketten zum Spiel.

zu Punkt 2

Das ist schon komplizierter, erst recht, wenn man direkt auf den Bildschirmspeicher zugreift und parallel dazu das VDI benutzt oder auf ein Monitor benutzt wird, der nur diese Auflösung benutzen kann (z.B. SM124) und man daher nicht umschalten kann. Da nutzt auch kein vt_trn_fm(), da dieser VDI Aufruf nicht dithert, also man in monochrom dann nicht mal annähernd erkennen kann, was es vorher mal für ein Bild war, was man da umgewandelt hat. Der vr_trn_fm() ist also nur brauchbar, um von Auflösungen mit weniger Bitplanes auf Auflösungen mit mehr zu konvertieren.

zu Punkt 3

Das sollte, wenn Punkt eins befolgt wurde, kein Problem mehr sein.

zu Punkt 4

Heißt soviel wie bitte keine Programme, "bei welchen man den Rechner mit einem Pflasterstein beschmeißen muß" damit er das Programm beendet. Also schön alles retten (Vektoren, Palette etc) und nach der Beendigung des Programms wieder zurückschreiben.

zu Punkt 5

Atari verkauft demnächst verbesserte Joysticks, die an die zusätzlichen Ports am 520/1040 STE/Falcon030 passen. Diese sollen demnächst vorrangig verwendet werden (mein Kommentar dazu folgt noch)

zu Punkt 7

Bisher habe ich noch keine Möglichkeit entdeckt, Action Spiele bzw. flackerfreie Animationen in einem Fenster unterzubringen (da müßte sich Atari noch etwas einfallen lassen) insofern gibt es nur wenige Arten von Spielen, die in ein "Fenster passen" z.B. Kartenspiele und einfache Adventures oder Spiele, bei denen ein bißchen Flackern bei der Animation nicht so tragisch ist.

zu Punkt 8

Spiele in Super - VGA Auflösung wären schon nicht schlecht, aber dazu braucht man entweder ein schnellere VDI (als das in den ROM's vom Falcon und allen anderen ATARIs) oder muß den Blitter des Falcon030 per Hand programmieren.

zu Punkt 9

Siehe, was ich zu Punkt zwei geschrieben habe.

Hier der Aufruf des vr_trn_fm() oder auch vr_trnfm(). Der Copy - Transform - Form ermöglicht es, das gerätezpezifische Format in ein Standard Format umzuwandeln und umgekehrt.

Deklaration in C:

```
void vr_trnfm(WORD handle, MFDB *psrcMFDB, MFDB *pdesMFBD)
{
    i_ptr (psrcMFDB);
    i_ptr2 (pdesMFDB);
    contrl [0] = 110;
    contrl [1] = contrl [3] = 0;
    contrl [6] = handle;
    vdi();
}
```

GEM - Felder:

contrl 110 Nummer von Transform Form
contrl+2 0
contrl+4 0
contrl+6 0
contrl+8 0
contrl+12 handle
contrl+14 Zeiger auf MFDB Quellraster
contrl+18 Zeiger auf MFDB Zielraster
Format der MFDB (memory definition Block):

16 Bytes mit den Funktionen

Langwort : Speicheradresse der Rasters
wenn 0, aktueller Bildschirm
Wort : Breite des Rasters
Wort : Höhe des Rasters
Wort : Größe des Bildschirmspeichers einer Plane für eine Zeile in Worten z.B. 20 Worte für ST - niedrig
Wort : 0 - Gerätespez./ 1 - Standard
Wort : Menge der Bitplanes z.B. 4 = 16 Farben

dann 3 reservierte Wörter

Das Standardformat ist planeweise organisiert. Das bedeutet, daß jede Bitplane komplett (ähnl. wie beim AMIGA der Bildschirmspeicher organisiert ist) in einem Stück hintereinander abgelegt ist, wobei die Organisation der Planes von links nach rechts d.h. 1. Plane Pixel 0,0 wird durch das 16. Bit des ersten Worts der Plane repräsentiert. In Monochrom stimmt das Standard Format mit dem geräteabhängigen Format (also so wie es im Bildschirmspeicher angeordnet ist) überein. Allerdings erlaubt es das VDI prinzipiell nicht, daß man irgendwelche Annahmen über das Format des Bildschimspeichers macht.

8.4.2 Mein Kommentar zu ATARIs Guidelines

Der Falcon030 ist kein Macintosh Rechner, er deckt ein komplett anderes Marktsegment ab. Dieses Marktsegment ist beherscht von Konsolen mit superschnellen Grafiken und PC's mit Rechnerleistung zu Dumpingpreise. Insofern muß ATARI, und nur der Hersteller kann dem Programmierer diese Möglichkeiten geben, eine Unterstützung durch das Betriebssystem bieten, Animationen laufen zu lassen und Spiele zu programmieren, die auch auf der nächst höhren Rechnergeneration laufen. Es darf einfach nicht passieren, daß auf dem TT die Spiele nicht laufen, nur weil ATARI vergessen hat, die Joystickports des 1040 STE einzubauen und bei der nächsten Rechnerfamilie vorschreibt: Jetzt nur noch mit den neuen Ports! Auch muß es möglich sein, die Spiele so zu programmieren, daß sie sowohl auf dem Falcon als auch auf einem ST mit Grafikkarte laufen. Da muß ATARI einfach eine Schnittstelle z.B. in Form eines META-Treibers liefern, die jeder Kartenhersteller auf seiner Karte implementieren "muß", die Sachen wie Page Flipping und/oder Synchronisieren der Grafikausgabe mit dem Bildaufbau, u.v.m. ermöglicht, um somit eine Standardisierung zu erreichen, die die Aufwärtskompatibilität von Animationsprogrammen, Spielen und Multimedia bei unter diesen Gesichtspunkten geschriebenen Programmen sicherstellt. Nur so, denke ich, ist ein Überleben dieser Rechnerfamilie, die unbestreitbar die Brücke zwischen PC/AMIGA und Macintosh/NEXT schlägt, möglich.

TEIL II

Live could be much easier,
if I had the source code

Autor: *unbekannt*

Was erreiche ich wie auf dem ATARI ?

Sie haben sicherlich schon Spiele und Demos gesehen, die auf dem Bildschirm Effekte zeigen, die sie den Atari Rechnern gar nicht zugetraut hätten und statt ohrenbetäubendem Piepsen erlesene Sounds "zum Besten gegeben haben". Vieles davon ist gar nicht so schwer zu programmieren, wenn man erst mal weiß wie !
Auch kann der interresierte Beobachter leicht erkennen, wie schlecht einige Programmierer, die Spiele von anderen Systemen, auf die ATARI Serie umgesetzt haben !

Dieser zweite Teil des Buches soll sowohl dem Hobbyprogrammierer, der guten Sound und weit mehr als 16 Farben in seine Programme integrieren will, das nötige Handwerkszeug bieten, als auch dem Profi, der nicht weiß, wie er diese oder jene "Spielerei" ohne Verluste auf den ATARI übertragen kann, helfen, die Resourcen der Ataris voll auszuschöpfen.
Nicht zuletzt soll gezeigt werden, daß man die Fähigkeiten der neuen ATARI Modelle nutzen kann ohne die alten ST's zu vernachlässigen.

B.1.Die Grafikhardware

der ATARI ST/STE/TT/Falcon030 Modelle

Als der ATARI ST Ende 1985 auf den Markt kam, sprach die Presse von hervorragenden Grafikmöglichkeiten; das ist auch nicht weiter verwunderlich, da die damals erhältlichen IBM PC kompatiblen Rechner nur mit CGA - Grafikkarten ausgerüstet waren, die bei einer Auflösung von 320*200 gerade mal 4 Faben und bei 640*200 nur zwei Farben darstellen können. Selbst damals gebräuchliche Homecomputer C64/C128/ATARI XL/Schneider CPC/ ZX Spectrum usw. waren kaum dazu fähig, den ATARI in Auflösung und Farbauswahl zu übertreffen. Nur der ein halbes Jahr später auf dem Markt erschienene AMIGA 1000, der eigentlich als Edelspielkonsole konzipiert war, übertraf in Sachen Grafik die ST Serie. Da der AMIGA 1000 in seinen Anfangszeiten mehr als doppelt so teuer war wie der 260 ST, war er keine direkte Konkurrenz. Das hat sich allerdings spätestens mit dem Erscheinen des AMIGA 500 geändert, der im Moment eher eine Konkurrenz zu den Spielkonsolen darstellt, er somit nach mehr als 5 Jahren endlich seiner Bestimmung gefolgt ist. Auch bringen heutige PC's mit VGA oder besser noch Super-VGA-Karten Grafiken, die man einem IBM-PC-Kompatiblen vor einigen Jahren gar nicht zugetraut hätte. Jedoch läßt sich mit einigen Tricks und Kniffen weitaus mehr aus dem ATARI herausholen als die Werte der Standardauflösungen vermuten lassen, zudem hat ATARI es mittlereile geschafft, auch Computer mit VGA ähnlichen oder noch besseren grafischen Möglichkeiten zu konstruieren.

1.1 Grafik Hardwareregister

Achtung !

Die Änderungen der Hardwareregister des Falcon werde ich am Ende dieses Kapitels extra beleuchten, weil Atari seine Politik alles zu dokumentieren zu Gunsten der Kompatiblität mit späteren Geräten geändert hat.

Darstellung der Register und deren Bedeutung:

in folgender Reihenfolge

Adresse: $XXXXXXXXX

Länge : L/W/B = 32/16/8 Bit

Bits : %XXXXXXXXX wobei
X - nicht belegt
n - belegt bedeutet

Zugriff : RO =Read only / nur lesbar
RW =Read and Write / beschreiben und lesen möglich
W =Write/beschreiben aber nicht immer lesen möglich

Label : Gebräuchliches Label in Symboltabellen von Debuggern und Programmen

1.1.1 Register und deren Funktionen

Videokontroller (Shifter)

Video Basis Register:

High - byte
$ff 8201 B %nnnnnnnn RW vbaseh

Mid - byte
$ff 8203 B %nnnnnnnn RW vbasemid

Low - Byte (nur STE/TT)
$ff 820c B STE %nnnnnnnX RW vbaselo
TT %nnnnnXXX

Das Beschreiben dieses Registers verändert die Speicheradresse von der der Videochip seine Bilddaten holt. Die Änderung wird erst beim nächsten Bildaufbau vom Shifter übernommen. Beim der ST Serie ist das die einzige Möglichkeit, den Bilschirmspeicher zu verschieben, auch läßt das Fehlen eines Registers für die unteren 8 Bit bei der ST Serie nur Bildschirmadressen an 256 Byte Grenzen zu, was wiederum das horizontale Scrolling erschwert; das wurde bei STE/TT geändert.
Bei STE läßt sich die Bildschirmadresse mit dem 'Low byte' auf Wort-Grenzen bringen, beim TT auf 64 Bit-Grenzen.

Video Adress Counter:

(Achtung wurde in einigen Büchern, im Bezug auf den TT, falsch beschrieben)

High - byte
$ff 8205 B %nnnnnnnn ST RO vcounthi
STE/TT RW
Mid - byte
$ff 8207 B %nnnnnnnn ST RO vcountmid
STE/TT RW
Low - byte
$FF 8209 B ST/STE %nnnnnnnX im ST RO sonst RW vcountlo
TT %nnnnnXXX STE/TT RW

Bei der ST-Serie kann man in diesen Registern nachlesen, an welcher Stelle im Bildschirmspeicher sich der Videochip gerade befindet. Das kann man z.B. dazu nutzen, um eine bestimmte Stelle abzupassen z.B. die Mitte einer Zeile oder überhaupt eine bestimmte Zeile (das Warten auf eine bestimmte Zeile geht mit Hilfe des Timers B viel eleganter als mit Hilfe dieser Register).

Beim STE ist es auch möglich, die Video Adresscounter zu (das erste Bit des 'Low - byte' wird ignoriert) beschreiben, mit ihnen ist es möglich, Split - Screen Darstellungen und (mit Hilfe zusätzlicher Register) auch voneinander unabhängig scrollende Streifen auf dem Bildschirm zu realisieren und das ohne größeren Rechenaufwand.

Beim TT sind diese Register ebenfalls zu beschreiben, allerdings werden die 3 ersten Bits des 'Low - byte' ignoriert. Das Ignorieren der unteren 3 Bits hat seinen Ursprung in der ausgeklügelten Video Hardware des TT. Um Timing Schwierigkeiten der CPU auf den Speicher zu umgehen, ist das Video - RAM des TT für den TT-Shifter 64! Bit breit, der Shifter liest also beim Zugriff auf das Video RAM immer volle 64 Bit und stellt diese dar.

In der Zwischenzeit darf die CPU auf den Speicher (ST - RAM) zugreifen, was laut den Unterlagen ein optimales Bustiming in einem Wechsel Shifter/Processor von 250 ns bedeuten würde, was wiederum 0 Waitstaits (Wartezeiten beim Zugriff auf das RAM) für einen TT mit 16 MHz 68030 (bzw. ca. >10% Wartezeit durch Überschneidungen CPU mit der VIDEO DMA) bedeutet hätte, allerdings wurden "in letzter Sekunde" alle TT mit einer 32 MHz-(Taktfrequenz, die die möglichen Taktzyklen pro Sekunde bestimmt) CPU ausgerüstet.

Beim Beschreiben der Video Adress Counter des TT, nimmt der TT die Änderung erst an, wenn er die letzten 64 bit angezeigt hat; das macht den Versuch, mitten in einer Zeile auf einen anderen Bildschirmspeicherbereich umzuschalten, etwas komplizierter.
ACHTUNG diese Eigenschaft des TT ist von ATARI nicht dokumentiert

Das Sync-Mode-Register

```
$ff 820a   B    ST/STE  %XXXXXXnn  RW  syncmode
                TT      %XXXXXXXn
```

Bit 0 ist für die Umschaltung auf externe Synchronisation zuständig, wenn es beim ST/STE gelöscht ist, wird der Video Prozessor intern synchronisiert (also ganz normal), wenn es gesetzt ist, extern (von außen) z.B. von einem Videorecorder für Genlock Anwendungen. Beim TT wurde die Bedeutung des Bits unglücklicherweise invertiert.

Bit 1 ist bei ST/STE für die 50/60 HZ Umschaltung zuständig, allerdings ist es mir bei einigen 1040 STE schon öfter passiert, daß sie bei der Umschaltung von 60 auf 50 Hz die Bitplanes nicht mehr ganz in richtiger Reihenfolge angezeigt haben. Dieses Bit erfüllt durch geschicktes Setzen und Löschen an bestimmten Stellen des Bilschirms noch einen anderen Zweck, denn mit ihm ist es möglich, sogenannte Fullscreen Darstellungen zu realisieren. Beim TT ist dieses Bit nicht belegt und wird folglich ignoriert (jedoch sollte man sich niemals darauf verlassen, daß bestimmte Bits in Registern auch weiterhin ignoriert werden).

ST kompatible Farbregister:

r=rot/g=grün/b=blau

Groß 'R' bzw. 'G' bzw. 'B' repräsentieren das niederwertige (das was am wenigsten ändert) Bit pro Nibble (4 bit) beim STE/TT.

```
$ff 8240  W ST    %XXXX Xrrr   Xggg Xbbb  RW     color0
            STE/TT %XXXX Bbbb   Gggg Rbbb
bis
$ff 825e                                         color15
```

liegen in Wortweise hintereinander !

Die Farbregister enthalten die Farbwerte aus den 512 bzw. 4096 Farben, die den 16 Farben z.B. in der niedrigen ST/STE Auflösung den 16 Farben zugeordnet werden. Änderungen der Farbregister machen sich auf dem Bildschirm sofort! bemerkbar, so daß man mit ein wenig Aufwand bis zu 48 Farben in einer Zeile darstellen kann und in jeder Zeile 46 neue Farben. Für die meisten Anwendungen reicht es jedoch, ein oder zwei Farbregister pro Zeile oder mal alle 16 Register mitten im Bild zu wechseln. Wie man sehen kann, sind beim ST von jedem Nibble (4Bit), das die Grundfarben R-rot G-grün B-blau im Farbregister repräsentieren nur 3 Bit genutzt, das macht 8-Bit Kombinationen pro Grundfarbe, was wiederum 8 hoch 3 Farben, also 512 Farben sind.

Beim STE wird zusätzlich noch ein niederwertiges Bit vom Shifter ausgewertet, dieses Bit liegt an ungewohnter Stelle, und zwar an dem jeweils 4. Bit der Farbnibbles. Die Palette des STE ist damit weitgehend ST kompatibel.

Die ST/STE kompatible Palette im TT ist im Prinzip ein Fenster aus der TT Palette, die 256 Farbregister beinhaltet. Bei der 256 Farbpalette sind sogar die Bits wieder in der richtigen Reihenfolge, d.h. das niederwertigste Bit ist das 1. Bit pro Nibble.

Das ST Shift-Mode-Register

$ff 8260 B	%XXXXXXnn		RW	shiftmd
Bitkombination	00	320*200	4 Bitplanes	16 Farben
	01	640*200	2	4
	10	640*400	1	Monochrom
	11	Reserviert		

Bewirkt die Auflösungsumschaltung! Dieses Register reagiert sofort! d.h. man kann z.B. die Auflösung mitten in einer Zeile (siehe GFA Raytracer) umschalten.

Das TT-Shift-Mode Register (nur TT)

$ff 8262 W %sXXh Xmmm XXXX pppp RW shift_tt

pppp (4 Bit) gibt das Fenster an, in das die ST Palette eingeblendet wird, die ST Palette läßt sich somit in 16er Schritten in der TT Palette verschieben. Bei einem Zugriff auf einer der beiden Paletten wird, falls es sich um ein Palettenregister handelt, das sich im hier eingestellten Fenster befindet, das Register in beiden Paletten geändert.

mmm (3 Bit) gibt die Auflösung an (änl. ST-Shift-Mode-Register)

Bitkombinationen:				
	000	320*200	4 Bitplanes	16 Farben
	001	640*200	2	4
	010	640*400	1	Duochrome
	011	Reserviert *		
	100	640*480	4	16
	010	Reserviert *		
	110	1280*960	1	Monochrome
	111	320*480	8	256 Farben

Achtung: auch die reservierten Auflösungen bringen ein Bild auf den Schirm !

Modus 3 (011), stellt 320*200 Punkte mit schätzungsweise 16 Farben dar, allerdings sind die Bitplanes so verdreht, daß ich mich da auch irren könnte ! einfach mal ausprobieren.

Modus 5 (010), hat 640*480 Punkten mit zwei Bitplanes, also 4 Farben, benötigt jedoch 4 Bitplanes, die Planes 3 und 4 werden ignoriert aber trotzdem vom Shifter eingelesen. Ich dachte zuerst, daß dieser Modus 1280*480 Punkte hätte, wegen seinem hohen Speicherbedarf - wäre schön gewesen !

Diese letzten beiden Modi sind von ATARI nicht dokumentiert, also nur was für Demo-Programmierer, die damit spezielle Effekte erzielen wollen. Diese zusätzlichen Auflösungen lassen sich nicht mit

den entsprechenden Xbios Aufrufen (5 - Setscreen) erreichen, da bei einem Aufruf dieser Auflösungen der Atari Bomben wirft, warscheinlich stürzen die Line A Variablen, VT 52 Emulation etc ab, da es für den Bildschirmtreiber unbekannte Modi sind.

h (1 Bit) schaltet auf den Hyper-Mono-Mode !
d.h. es besteht eine Auswahl aus 256 Graustufen und nicht mehr aus 4096 Farben, wobei das blaue Nibble in den Farbregistern die unteren 4 Bit und das grün Nibble die oberen 4 Bit der 8 Bit Graustufen Palette sind. Die 4 Bit, die für den Rot-Wert im Palettenregistern zuständig sind, werden in diesem Modus ignoriert. Damit ist im 320*480*8 Mode eine echte Graustufen-Darstellung möglich !

s (1 Bit) Sample & Hold-Mode
Beim Setzen eines Pixels mit einer Farbe ungleich der Hintergrundfarbe, wird diese bis zu Ende der Zeile aufgefüllt, falls nicht eine andere Farbe zwischen dem ersten gesetzten Pixel und dem Rand gesetzt wird, sonst wird diese bis zum Rand gezeichnet. Dabei wird nicht der Bilschirmspeicher verändert sondern der Videoprozessor "merkt" sich jeden Wert einer Zeile, die ungleich der Hintergrundfarbe ist und stellt diesen dar, bis er auf eine andere Farbe trifft, die er sich merken kann. Jedoch "merkt" sich der Videoprozessor die Farben nur bis zum Ende einer Zeile.

TT Farbpalettenregister (nur TT)

r=rot / g=grün / b=blau

$ff 8400 W %XXXX rrrr gggg bbbb RW TT_col0
bis
$ff 85fe W %XXXX rrrr gggg bbbb RW TT_col255

groß ist die CLUT (Color-Look-Up-Table) des TT, sein Inhalt wirkt sich sofort nach dem Beschreiben auf den Bilschirm aus. Wenn der Hyper-Mono-Mode aktiviert ist, wird das Rot-Nibble ignoriert.

Register zum Hardwarescrolling (nur STE/F030)

Pixel-Offset
$ff 8256 B %XXXX nnnn RW hscroll

Zeilenbreite
$ff 820f B %nnnn nnnn RW linewid

Mit diesen beiden Registern, in Verbindung mit dem 'vbaselo', ist es möglich, virtuelle Bildschirme zu erzeugen, daß sind große Bildschirme, die sich im Speicher befinden von denen nur ein Ausschnitt auf dem Monitor dargestellt werden kann. Dabei kann dann mit Hilfe dieser Register der Ausschnitt fast ohne Verschwendung jeglicher Rechenzeit durch kopieren von Bildschirmausschnitten, der sichtbare Bildschirm im Virtuellen, in 1 (bei Bedarf auch mehr) Pixel Schritten, gescrollt werden.

Bei allen von mir getesteten STE's funktionierte das 'hscroll' Register nur in der niedrigen Auflösung einwandfrei, in der mittleren und hohen Auflösung erzeugte das Zurücksetzen des Registers von irgendeinem Wert auf Null ein Monitorbild, bei dem die Zeile in den Zeilenrücklauf hineinreichte. Man kann diesen Fehler nur umgehen, wenn man in diesen beiden Auflösungen nur Werte ungleich Null in das 'hscroll' Register schreibt.

1.1.1 Betriebssystem Funktionen zum erreichen der Videoharware

Mein Dank geht an dieser Stelle an Herrn Kowalewski von ATARI Deutschland, für das flotte Zusenden der Falcon030 Entwickler-Unterlagen und Software.

Nomenklatur:

WORD = 16 Bit
LONG = 32 Bit
VOID = Dummy, d.h. keine Rückgabe
* = Zeiger auf

Beispiel:

XBIOS 5

VOID Setscreen(LONG log, LONG phys, WORD rez, WORD mode)

ist

```
    move.w      #mode,-(sp)
    move.w      #rez ,-(sp)
    move.l      #phys,-(sp)
    move.l      #log ,-(sp)
    move.w      #5   ,-(sp)
    trap        #14
    lea         $e(sp),sp
```

Rückgabe erfolgt gegebenenfalls ins Register D0.

XBIOS 2

LONG Physbase()

Gibt Adresse des momentan angezeigten Bildschirms zurück !

XBIOS 3

LONG Logbase()

Gibt Adresse des momentan vom Grafiktreiber angenommenen Bildschirmspeichers zurück !

XBIOS 4 (nur ST(E)/TT Auflösungen)

WORD Getrez()

Gibt den momentanen Grafikmodus zurück (funktioniert beim FALCON030 nur in den ST kompatiblen Modi einwandfrei)

XBIOS 5

VOID Setscreen(LONG log, LONG phys, WORD rez, WORD mode)

Zum Setzen von:

phys , des anzuzeigenden Bildschirmspeichers (siehe XBIOS 2)
log , der zu beschreibenden Bildschirmspeicher (siehe XBIOS 3)
rez , Bildschirmauflösung ST(E)/TT Auflösungen keine voll TT kompatiblen Auflösungen auf dem Falcon030 möglich.
mode , F030 Auflösungen (siehe XBIOS(88))

-1 für einen der Parameter bedeutet keine Änderung

XBIOS 6 (nur ST(E) und TT bis 16 Farb Auflösung)

VOID Setpalette(LONG *Palette)

*Palette, Zeiger auf ein Array mit 16 Wörtern, die die Werte für 16 Farbregister enthalten. 1 WORT hat folgendes Format:

ST	%XXXX Xrrr	Xggg	Xbbb
STE/TT	%XXXX Bbbb	Gggg	Rbbb

r=rot, g=grün, b=blau
RGB = niederwertigstes BIT rot/grün/blau

Wird erst beim nächsten VBL gesetzt

XBIOS 7 (nur in ST(E) und TT bis 16 Farb Auflösungen)

WORD Setcolor(word colornum, word color)

colornum , (0..15) Nummer des Palettenregisters
color , siehe XBIOS 6

gibt bisherigen Wert des Farbregisters zurück.

XBIOS 80 (nur TT)

WORD EsetShift(WORD shiftMode)

shiftMode

Bit:	0..3	Nummer des Farbregisters
	8..10	Auflösung
	12	Wenn gesetzt, Hypermono (256 Graustufen)
	15	Smear Modus

gibt alten Wert zurück.

XBIOS 81 (nur TT)

WORD EgetShift()

gibt ein 16 Bit zurück, wobei:

Bit:		
	0..3	Nummer des Farbregisters
	8..10	Auflösung
	12	Wenn gesetzt, Hypermono (256 Graustufen)
	15	Smear Modus

siehe auch $ffff8262 shift_tt Kapitel B.1.1.1.

XBIOS 82 (nur TT)

WORD EsetBank(WORD banknum)

banknum, Nummer des benutzten Fensters in der TT 256 Farb-Palette (16 Register weise), bei negativer 'banknum' erfolgt keine Änderung.

gibt alte 'banknum' zurück

XBIOS 83 (nur TT)

WORD EsetColor(WORD colornum, WORD color)

colornum,	wie XBIOS 7 nur komplette 256 Farben
color ,	%XXXX rrrr gggg bbbb
	wenn negativ - keine Änderung

gibt alten Wert zurück.

XBIOS 84 (nur TT)

VOID EsetPalette(WORD colorNum, Word count, LONG *palette)

colorNum , Nummer des Startregisters
count , Menge der zu lesenden Register
*palette , Zeiger auf ein Feld mit 4 Bit RGB Werten
%XXXX rrrr gggg bbbb

kopiert die Farbwerte im mit '*palette' angegebenen Feld in die Farbregister

XBIOS 85 (nur TT)

VOID EgetPalette(WORD colorNum, WORD cont, LONG *palette)

colorNum , Nummer des Startregisters
count , Menge der zu lesenden Register
*palette , Zeiger auf ein leeres Feld

schreibt Inhalt der Paletteregister in den angegebenen Speicherbereich.

XBIOS 86 (nur TT)

WORD EsetGrey(WORD switch)

switch, wenn >0 schaltet auf Hypermono
wenn =0 schaltet auf 4096 Farbpalette zurück
wenn <0 Register wird nicht geändert

siehe auch $ffff8262 shift_tt Kapitel B.1.1.1

gibt alten Wert zurück.

XBIOS 87 (nur TT)

WORD EsetSmear(WORD switch)

switch, wenn >0 schaltet in den Sample and Hold Modus
wenn =0 schaltet zurück in den "normalen Modus"
wenn <0 Register wird nicht verändert

siehe auch $ffff8262 shift_tt Kapitel B.1.1.1

gibt alten Wert zurück.

XBIOS 88 (nur FALCON)

WORD Vsetmode(WORD modecode)

modecode, %XXXXXXXFSOPV8NNN

NNN 0 = 1 Bits per Pixel, Mono/Duochrome
1 = 2 Bits per Pixel, 4 Farben
2 = 4 Bits per Pixel, 16 Farben
3 = 8 Bits per Pixel, 256 Farben
4 = 16 Bits per Pixel, 65536 Farben

8 wenn gesetzt 80 Zeichen pro Zeile, sonst 40
V wenn gesetzt VGA Monitor, sonst Fernseher
P wenn gesetzt PAL 50 HZ, sonst NTSC 60 HZ
O wenn gesetzt X,Y Auflösung mal 1.2
S wenn gesetzt ST kompatible Auflösung/Palette etc.
F Interlace an, wenn P gesetzt
X reserviert für zukünftige Erweiterungen

Nicht unterstützt wird das Mischen von:

80 Zeichen pro Zeile also 640 Spalten und VGA bei 16 Bit p. P.
40 Zeichen pro Zeile also 320 Spalten und 1 Bit p. P.

wenn modecode=-1 keine Änderung der Auflösung, gibt alte Auflösung zurück

ATARI empfielt 'Setscreen' zum Ändern der Auflösung, da 'Vsetmode' das VDI nicht umschaltet.

XBIOS 89 (nur FALCON)

WORD mon_type()

Gibt zurück, welcher Monitor am Falcon angeschlossen wurde.

0 = ST Monochrome Monitor
1 = ST Farb Monitor
2 = VGA Monitor
3 = Fernseher

XBIOS 90 (nur FALCON)

VOID VsetSync(WORD external)

external, %XXXXXXXX XXXXXhvc

wenn gesetzt

h = externe horizontale Syncronisation
v = externe vertikale Syncronisation
c = externe Pixelfrequenz

Wird zur Synchronisation mit Genlocks u. Videobandmaschinen etc. verwendet.

XBIOS 91 (nur FALCON)

LONG Vgetsize(WORD mode)

mode, siehe XBIOS 88

gibt den Speicherbedarf der in mode angegeben Auflösung zurück

XBIOS 93 (nur FALCON)

VOID VsetRGB(WORD index, WORD count, LONG *array)

index , Farbregister Start
count , Menge der zu übertragenen Register
*array , n Wörter mit dem Paletteninhalt "Xrgb"

X = unbenutzt %XXXXXXXX
r = ein Byte Intensität rot %rrrrrrXX g = ein Byte Intensität grün %ggggggXX
b = ein Byte Intensität blau %bbbbbbXX

Überträgt Palette aus dem Speicher in die Palettenregister.

XBIOS 94 (nur FALCON)

VOID VgetRGB(WORD index, WORD count, LONG *array)

index , Farbregister Start
count , Menge der zu übertragenen Register
*array , leeres Feld

überträgt Palette aus Palettenregister in den Speicher.

XBIOS 150 (nur FALCON)

VOID VsetMask(WORD andmask, WORD ormask)

andmask , Maske für AND Verknüpfung (default $FFFF)
ormask , Maske für OR Verknüpfung (default $0000)

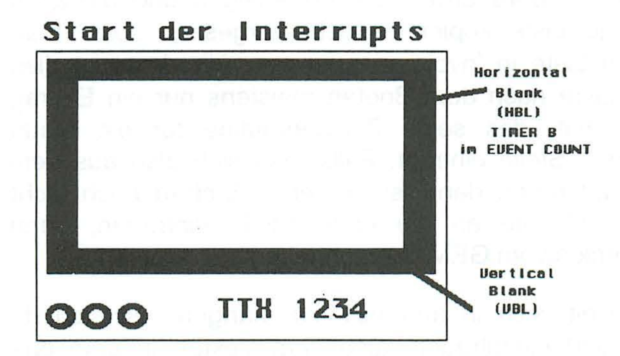
Darf nur in "True" - Color - Modi verwendet werden !

Diese Masken werden mit den durch vs_color (VDI) gesetzten Farben verknüpft um die Möglichkeit zu bieten, auch über das VDI gesetzte Bildpunkte bei der Benutzung eines Genlocks durchsichtig erscheinen zu lassen.

1.2 Hardware Scrolling auf dem STE

Ein großes Manko der Atari ST Computer bei der Spieleprogrammierung ist wohl das nur sehr umständlich zu realisierende horizontale Scrolling. Scrolling, für alle die dies bis jetzt immer noch nicht wissen, ist das Hin und Her-Bewegen des Hintergrundes bzw. der Spielfläche. Das beste Beispiel für butterweiches 8 Wege Scrolling ist der Gauntlet Automat von Atari, aber auch viele Spiele/Demo Programmierer haben schon recht gutes Scrolling auf einem Standard ST zustande gebracht. Da aber das Bewegen riesiger Bildschrimblöcke entweder 'ne Menge Speicher oder 'ne Menge Rechenzeit (nicht jeder hat einen 68030 oder einen Blitter) schluckt, hat Atari den Videoprozessor in den STE's so modifiziert, daß man sich einfach einen großen Bildschirm im Speicher anlegt, in dem dann der auf dem Monitor angezeigte Bildschirm nur einen Ausschnitt darstellt, der nach Belieben in dem im Speicher befindlichen hin und her geschoben werden kann. Dies geschiet durch das Verändern einiger weniger Register; der Aufwand hierfür fällt dann effektiv nicht mehr ins Gewicht.

1.2.1 Der VBL (Vertikal Blank) Interrupt



Nun, wird mancher sich fragen, was hat den ein Interupt mit Scrolling zu tun ?
Es ist im Prinzip derselbe Grund wie beim Bewegen von Sprites, denn die Bildausgabe muß eben mit dem Bildaufbau synchronisiert werden, und da die Register, die zur Vereinfachung des Scrollings dazu gekommen sind, sofort reagieren, dürfen sie erst vor dem Aufbau des Monitorbilds beschrieben werden. Genau an diesem Punkt (nach der letzten dargestellten Zeile) erzeugt eben die Videohardware des Atari einen Interrupt .Bedingt durch den 68000er, der in allen ST's zu finden ist, gehen alle Programmierer davon aus, daß die Vektoradresse zum VBL auf $70 befindet. Bei den neueren Prozessoren 68010/20/30/40/60 läßt sich jedoch die Vektortabelle, in der sich der VBL befindet, beliebig im RAM verschieben. Es ist jedoch nicht zu erwarten (allein von der Sytemarchitektur), daß sich die Vektortabelle in kommenden TOS-Versionen verschieben wird. Jedoch ist es möglich, mit 'setexec' (BIOS 5) die Vektorenbetriebssystemkonform zu ändern.

Nun gibt es mehrere Methoden, sich in den VBL Interrupt zu "hängen". Eine saubere Methode ist es, die Adresse der eigene Routine, die während des VBL aufgerufen werden soll, in eine Liste von anderen Adressen zu schreiben. Diese Liste wird automatisch von der VBL Routine des Betriebssystems ausgeführt. Die Systemvariable '_vblqueue' an der Adresse $454 zeigt auf diese Liste . An der Adresse $454 'nvbls' ist dann die Größe der Liste eingetragen

(normalerweise eine acht). Sollten alle Einträge belegt (also <>0) sein, sollte man eine größere Liste im RAM anlegen und die alten Adressen in die neue Liste kopieren. Nicht vergessen sollte man die Größe der neuen Liste in 'nvbls' einzutragen. Allerdings ist dies selten nötig, da die Liste nach dem Booten meistens nur ein Eintrag vom Betriebssystem hat, das seine Zeichenroutine für die Maus immer (!) an die erste Stelle einträgt. Falls man sich also aus dem 'AUTO' Ordner einträgt (denn dann ist der erste Eintrag noch nicht belegt), sollte man sich nie an die erste Stelle eintragen, denn sonst wird man gnadenlos vom GEM überschrieben.

Eine andere Möglichkeit, sich in den VBL zu "hängen", ist sich direkt in die Adresse $70 einzutragen, also dem Vektor, in dem das Betriebssystem "hängt" und nach Beendigung der eigenen Routine wieder das Betriebssystem anzuspringen. Das hat wesentliche Vorteile! Denn man kommt als erster zum Zug, was sehr wichtig ist, wenn man viel während des VBL's zu erledigen hat und jede Verzögerung auf dem Bildschirm zu sehen wäre. Auch ist diese Methode mit geringem Aufwand zu realisieren. Wenn es sich bei dem Programm, das man in den Bildrücklauf installiert, um ein residentes Programm handelt oder um ein Programm, das in einem Fenster auf dem Desktop läuft (also man sich möglicherweise nicht allein im Rechner befindet), sollte man das XBRA Verfahren benutzen, da dieses Verfahren es möglich macht, sich gegebenfalls wieder aus dem Vektor zu entfernen oder zumindest sich nicht ein zweites Mal zu installieren. Der VBL-Interrupt wird immer dann aufgerufen, wenn sich der Elektronenstrahl des Monitors auf dem Weg vom unteren Bildschirmrand zum oberen befindet, also 50/60 oder ca. 71 mal pro Sekunde, es ist in dieser Zeit möglich, alle Farbregister zu verändern oder eine begrenzte (begrenzt durch die zur Verfügung stehende Rechenzeit) Menge von Sprites auf den Bildschirm zu bringen oder eben die Register zum Hardwarescrolling zu setzen.

1.2.2 Wie geht nun aber das Hardware Scrolling ?

Nun, Atari hat dem Shifter des STE/F030 einige neue Register spendiert, die das Scrolling fast zum Kinderspiel machen.

1.2.2.1 Vertikales Scrolling (auch auf dem TT möglich)

Um vertikales Scrolling zu vereinfachen, wurde dem Programmierer die Möglichkeit gegeben, mit Hilfe eines Low - Bytes ($ff 8209) die Bildschirmadresse wortweise im Speicher zu verschieben. Das ist eigentlich schon "die halbe Miete", denn dann braucht man nur zwei Bildschirme hintereinander in den Speicher zu laden und die Bildschirmadresse ganz langsam im Speicher herunter zu schieben. (auf der beiliegenden Diskette befindet sich im Programm 'kapitel1.TOS' das File 'stescroll.tos', das horizontales und vertikales Hardwarescrolling und gescrollte Teilbilder (Split Screen) zeigt.) Nun muß dies nicht unbedingt durch einen direkten Zugriff auf die Adresse geschehen! Es reicht vollkommen, die Betriebssystem-Routine 'Setscreen' (Xbios 5) zu benutzen, da diese an die neue Hardware angepaßt ist.

1.2.2.2 Horizontales Scrolling

Vertikales Scrollen läßt sich noch leicht ohne zusätzliche Hardware realisieren (auch wenn es mit viel leichter geht), jedoch das horizontale Scrolling ist auf einem normalen ST Sysiphus Arbeit. Daher gibt es für diesen Zweck zwei neue Register. Zum einen das 'Pixel Offset' Register, das den Videoprozessor dazu bewegen kann bis zu 15 Pixel (also fast das erste Wort) am Anfang der Zeile zu schlucken. Allerding würde bei einem Inhalt dieses Registers <> 0 die Zeile um den Inhalt des Registers länger werden. Um das auszugleichen, gibt es das 'Zeilenbreite'-Register, das den Prozessor dazu bringt, am Ende jeder Zeile bis zu 16 Worte im Bildschirmspeicher zu überspringen. Um z.B. zwei Monochrom-Bilder nebeneinander auf dem Bildschirm horizontal scrollen zu können, muß man das 'linewid' Register auf 39 setzen, für die mittlere Auflösung (und entsprechenden 4 vierfarbigen Bildern) 78 Worte und für die niedrige Auflösung 76 Worte. Eine leicht zu merkende Formel ist:

l =	länge einer Zeile in Worten			
	mittlere & niedrige ST Auflösung je 80 Worte			
	hohe ST Auflösung 40 Worte			
	Falcon, verschieden !			
b =	Menge der zu scrollenden Bilder nebeneinander (in einer Reihe)			
w =	Worte pro 16 Pixel			
	monocrome		Auflösung	1 Wort
	4	-farbige	Auflösung	2 Worte
	16	-farbige	Auflösung	4 Worte
	256	-farbige	Auflösung	8 Worte

'linewid' = l*(b-1)-w

Achtung ! 'linewid' kann nicht größer als 255 werden, da das Register nur ein Byte breit ist.

Aber nun mehr zum Praktischen !

Zuerst kopiere ich mir die zu scrollenden Bilder Zeile für Zeile hintereinander, z.B. bei zwei Bildern

erstes Bild erste Zeile / zweites Bild erste Zeile
erstes Bild zweite Zeile / zweites Bild zweite Zeile
. . . .
. . . .
erstes Bild nte Zeile / zweites Bild nte Zeile

Dann berechne ich mir den Wert für das 'linewid' Registers nach der oben aufgeführten Formel, setze das 'linewid' Register und auf dem Bildschirm erscheint (falls sich die Bilder im Bildschirmspeicher befinden) das erste Bild. Um dann zu scrollen, erhöhe ich das 'pixel offset' Register bis es den Wert 15 erreicht, dann setze ich dieses Register wieder auf Null, dann wiederum erhöhe ich die Bildschirmadresse um ein Wort und beginne von vorn. Wenn man diesen Vorgang nicht mit dem Bildrücklauf synchronisiert, sieht man das Umschalten der Register auf dem Monitor. Um das zu vermeiden, habe ich eine Routine geschrieben (als GFA-Basic-Inli-

ne), die diesen Zweck erfüllt. Um zu vermeiden, daß das Setzen der Register (wenn sich viele Programme in den Interrupt hängen, kann das passieren) zu spät erfolgt, hängt sich mein Programm als allererstes in den VBL Vektor, und springt dann die Routine an, die zuvor im Vertical Blank Interrupt (VBL) hing. Da der Video Prozessor den Inhalt der Video Basis Register möglicherweise schon in die Adress Counter übertragen hat (was bei jedem 16 Pixel ein leichtes Flackern bedeuten könnte), beschreiben wir an dieser Stelle auch gleich die Address Counter Register mit der Bildschirmadresse mit Offset (low byte) für's horizontale Scrolling, da der STE-Video-Shifter sofort auf das Beschreiben dieser Register reagiert. Viel Spaß !

1.2.3 Virtuelle Bildschirme

Es gibt mehrere Möglichkeiten, virtuelle Bildschirme zu realisieren. Allen zu Grunde liegt eine logische Erweiterung der Bildschirmauflösung, ohne physikalisch eine solche darstellen zu können. Im Klartext z.B. die Simulation eines 1280*960 Punkte großen Schirms (also TT hoch) auf einem SM 124 (also ST hoch mit 640*400 Punkten).

Diese Simulation ist zum einen ein Hardware-Problem, da man ein System finden muß, um den kleinen physikalischen Bildschirm im größeren logischen Bildschirm zu verschieben. Nehmen wir an, bei dem Rechner handele es sich um einen STE, dann ist das Problem mit Hilfe des Hardwarescrollings (siehe 1.2.1.2) zu lösen.

Dann bleibt immer noch das Problem, der Software beizubringen, mit einem größeren Bildschirm zu arbeiten. Die wohl "kompatibelste" Möglichkeit, mit AES & VDI die neue Auflösung zu nutzen, ist, einen eigenen GDOS Bildschirmtreiber zu schreiben. Nun, das ist erstens sehr aufwendig und zweitens wird diese Technik von keinem mir bekannten Programm zum Erzeugen virtueller Auflösungen genutzt. Wer nur dem internen Gerätetreiber des VDI beibringen möchte, daß sich "plötzlich" die Auflösung geändert hat, kann das mit Hilfe der sogenannten negativen Line A Variablen tun . Die

negativen Line A Variablen sind so etwas wie die lokalen Variablen des VDI. Beim Ändern dieser Variablen paßt man damit automatisch das VDI an. Allerdings hat davon das AES (der Teil des Betriebssystems, der für die Ausgabe von Menüs, Fenstern, Dialogboxen etc. zuständig ist) noch gar keine Ahnung, um das zu ändern, müßte man (oder auch Frau) das Programm in den Auto Ordner plazieren, also vor der Initialisierung von VDI und AES. Von dort ändert es dann die negativen Line A Variablen und setzt nach dem Öffnen der Workstation die richtigen Auflösungswerte in work_out(0) und work_out(1) ein. Aber die Anpassung von AES ist für den Spieleprogrammierer nicht allzu wichtig, da er es nur in den seltensten Fällen benutzt (z.B. bei Grafik Adventures oder Brettspielen) und dann meistens keinen virtuellen Bildschirm benötigt. Es ist aber gelegentlich nicht zu verachten, auf große zu scrollende Bilder der Einfachheit halber mit dem VDI zuzugreifen (z.B. für ein Zeichenprogramm oder eine Simulation).

1.3 Eine Auswahl von fast 4096 (3375) Farben auf eine normalen ST

Bei der Konvertierung vom Amiga auf den Atari haben die Programmierer oft das Problem der zu geringen Farbauswahl. Sie würden zwar mit den 16 Farben auskommen (indem sie mehrmals umschalten) aber die Palette der "alten" ST's ist mit 512 Farben einfach zu klein. Auch bei digitalisierten Bildern tritt häufig das Problem auf, daß man gerne mehr als 8 Graustufen auf dem Schirm darstellen möchte, da das viel hübscher ist als z.B. 16 Rot-oder Grün-Stufen.

Ich stelle hiermit eine Lösung für diese und ähnliche Probleme vor. Ich schreibe einfach ein Programm, das bei jedem VBL alle 16 Frabregister des ST wechselt, und dabei an den Stellen wo man eine Farbe erzeugen möchte, die sich nicht in den 512 Farben des ST befindet, wechselt man bei jedem VBL den Inhalt des Palettenregisters mit den Farben, die gerade noch darstellbar sind und oberhalb bzw. unterhalb der Wunschfarbe liegen. Dabei wird zwar ein leichtes Flackern sichtbar, allerdings nur bei größeren Flächen.

Wie funktioniert es genau ?

Also der ATARI ST kann normalerweise 16 aus 512 Farben darstellen. Wobei jede der 16 Farben ein Palettenregister besitzt. Diese Palettenregister besteht aus 4 * 4 Bit

R=rot G=grün B=blau U=unbenutzt

Palettenregister: (16 Bit also 1 Wort)
UUUU URRR UGGG UBBB

Wie man sieht, sind in den niederwertigen 12 Bit die RGB Informationen enthalten, wobei das 4. Bit pro Nible (4 Bit Block) bei ST's (nicht STE/TT) unbenutzt ist.

Das ergibt 3 Grundfarben mit je 3 Bit also 8 Stufen !

das ergibt 8 hoch 3 = 512 Farben

wenn man jedoch die Trägheit des Auges ausnützt und durch schnelles Wechseln der Farben für je eine Grundfarbe - RGB eine Zwischenstufe erzeugt.

Also etwa so:

z.B. ROT

```
0    1    2    3    4    5    6    7        Farbstufen
-------------------------------------------------------
  0    1    2    3    4    5    6          1. Farbe
  1    2    3    4    5    6    7          2. Farbe
-------------------------------------------------------
I I  I I  I I  I I  I I  I I  I I  I        effektive
1 2  3 4  5 6  7 8  9 1011 12 13 14 15      Farbstufen
```

Mir ist bisher nur ein Spiel bekannt, das diese Technik benutzt um mehr Farbabstufungen darstellen zu können, und zwar First Samurai von Imageworks. Auch wurde in der Ausgabe CT 6/92 ein Programm vorgestellt, das mit dem gleichen Verfahren die ST-Hoch Auflösung mit einem Grauton versieht, allerdings werden in diesem Fall zwei Bildschirmspeicher benutzt, die durch beschreiben der Video Basis Register immer umgeschaltet werden. Auch in anderen Auflösungen sind mit dieser Technik erstaunliche Effekte möglich, das Flackern läßt sich leider nicht vermeiden, nur durch eine 60 Hz Umschaltung erträglicher machen. Unter den Programmen zum Thema Bildformate befindet sich ein komplett dokumentiertes Listing, das IFF's Bilder lädt und auf einem ST die 4096 Farben des STE mit der von mir oben vorgestellten Technik simuliert.

1.4 Fullscreen und Syncscroller Programmierung

(nur bei ST/STE und in Farbauflösungen möglich)

In Demos und mittlerweile auch in einigen Spielen kann man Syncscroller (meist nur in Demos) und geöffnete Bildschirmränder bewundern (z.B. die Anfangsdemo zu Turrican II). Wenn man sich die Register des Ur-ST's anschaut, sind solche Effeckte nicht zu erklären, da keines dieser Register auch nur entfernt etwas mit Scrolling oder den Rändern des Bildschirms zu tun hat.

In den Scrolltexten einiger Demos liest man von einem gewissen "Alyssa", der als erster Programmierer den ST dazu gebracht hat Pixel, außerhalb seiner Bildbegrenzungen darzustellen. Alyssa öffnete den unteren Rand. Die erste Demo, die unteren Rand benutzte, müßte die "TEX and Alyssa Super - Neo Demo" gewesen sein, die im unteren Rand eine Laufschrift und ein bißchen Animation zeigt, etwas Musik spielt und Bilder im 'NEOCROME' Format in den oberen Teil des Bildschimspeichers lädt.

Danach folgte die 'B.I.G. D.E.M.O' (auch von TEX - also The EXceptions). Nachdem die Demo-Programmierer noch den oberen Rand öffneten, folgte auch bald die "Eröffnung" des rechten und linken Rands und nach einiger Zeit auch ein (Amiga-like) Fullscreen, der wohl zu allererst in dem 'LEVEL 16' Screen in der Union Demo zu sehen war, am unteren Rand des "main menue" Screen der Demo konnte man auch schon sogenanntes "Syncscrolling" bewundern.

1.4.1 Geöffnete Bildschirmränder

Das Heranschaffen, Darstellen und Syncronisieren von Videodaten wird im ATARI von drei Prozessoren übernommen. Zum einem der GLUE, der außer seiner vielen anderen Tätigkeiten, das HSYNC, VSYNC und das DE (Display Enable) Signal erzeugt. HSYNC und VSYNC sind für die Syncronisation des Monitors zuständig und das DE Signal, das auch an den Eingang vom Timer B geht, wird dann gegeben , wenn eine Zeile geschrieben wird (aber dazu komme ich noch im Kapitel über den Timer), und genau dieses Signal ist wichtig. In der MMU liegen die Video - Adress Counter; dieser Chip liefert die darzustellenden Adressen an den Shifter. Der Shifter wiederum erzeugt die RGB Signale für das Bild aus den Werten, die in seinen Palettenregister stehen. An den Rändern liefert der GLUE kein DE Signal, das den Shifter dazu bringt, nur noch Hintergrundfarbe (Palettenregister 0) dazustellen. Wenn man allerdings die Register des GLUE im richtigen Augenblick mit den richtigen Werten beschreibt, vergißt er das Löschen des Signals und der Shifter dekodiert munter Speicher zu RGB Signalen, was mehr Bild auf dem Monitor bedeutet. Aber nur, da die MMU, beim Fehlen des Display Enable, weiterhin Informationen an den Shifter liefert. Das Problem dabei ist nun die Toleranzen der drei Chips, je nach dem aus welcher Serie sie stammen kann man sie (per Software) zu mehr oder weniger Zeilen und Pixel überreden. Auch treten zwischen den ST und STE Chips größere Toleranzen auf.
Das ganze ist eben eine undokumentierte Eigenschaft der Bausteine, für die Atari nicht garantiert.

1.4.1.1 Das Öffnen des unteren Randes

(Was wohl die leichteste Disziplin ist !)

Im Prinzip muß man nur bis zur Zeile 199 warten, dann mal kurz von 50 auf 60 Hz schalten, etwas warten und dann wieder von 60 auf 50 Hz schalten, und schon hat man etwa 45 Zeilen mehr am unteren Bildschirmrand. Das komplizierteste dürfte das Warten auf die zweihundertste Zeile sein !

Hier der Ausschnitt eines Listings, zum realisieren einer relativ sauberen Umschaltung:

Routine wird vom Timer B an Zeile 198 gestartet, der im VBL -Interrupt darauf programmiert wird.

Timer B steht im 'EVENT COUNT' Modus (siehe Kapitel über Timer Programmierung).

```
timerb:
        move.l  D0,-(SP)                    ;wird nur in einem Fall benötigt
                                            ;Timer B zählt von Zeile 190 abwärts
warten:
        cmpi.b  #180,tbdr                   ;Wartet bis Timer B Data Register auf
        bne.s   warten                      ;180 ist, denn 190-180=10 ! also
                                            ;befindet sich der Rechner an der
                                            ;200sten Zeile
        andi.b  #11111101,syncmode          ;auf 60Hz schalten
```

Hier kommen jetzt zwei Möglichkeiten, auf den richtigen Moment zum Umschalten zu warten:

```
**********************************************
        move.w  #1,D0
pause:  nop             ;etwas
        dbra.s  D0,pause ;warten
**********************************************
```

oder besser noch (da kompatibler zu 68010 oder Beschleunigern)

```
**********************************************
        rept 14     ;macht 14 NOP's
        nop
        endr
**********************************************

        ori.b      #2,syncmode      ;wieder 50Hz
        move.b     #0,tbcr          ;Timer aus
        bclr       #0,isra          ;Interupt beendet, Interupt in
                                    ;service Bit löschen !!!
        move.l (sp)+,d0             ;d0 wieder zurück
        rte
```

Natürlich wäre auch ein 'MOVE' zum Syncmode Register möglich, würde aber auf dem TT zu einem schwarzen Bildschirm führen, da auf dem TT das Bit zur externen Syncronisation invertiert ist !!!

Diese Routine funktioniert nur auf 8 MHZ oder >16MHZ ohne Cache (nur die Routine mit den NOP's). Auf dem TT bleibt die 50/60 Hz Umschaltung ohne Wirkung, da bei diesem Gerät das dazugehörige Bit des 'Syncmode' Registers gar nicht existiert.

1.4.1.2 Das öffnen des oberen Randes

Etwas vorab, der HBL !

Der HBL (Horizontal BLank) wird theoretisch nach jeder Zeile ausgelöst, allerdings ist die Interruptmaske beim ST so hoch gesetzt, das der HBL nicht ausgeführt wird. Falls er doch ausgeführt wird, ab Blitter TOS nach jedem Programmstart, setzt die Routine, auf die der HBL Vektor zeigt, die Interuptmaske wieder so hoch, daß kein HBL mehr auftritt. Er liegt an der Adresse $68 (Offset zum Anfang der Vektortabelle - auf dem ST mit 68000 liegt diese ab Adresse 0 im Speicher $68 ist hier eine absolute Adresse) und ist ein Autovektorinterrupt, das heißt, es wird kein Interruptcontroller (z.B. MFP im ATARI) benötigt, um den Interrupt auszulösen.

Laut einiger Demo-Programmierer hängt die Menge der zu öffnenden Zeilen des oberen Bildschirmrandes von der im Gerät verwendeten MMU (Memory Management Unit), die die Video Adress Counter enthält, ab. Aber im Prinzip kann es dem Demo-Programmierer egal sein, welcher Baustein für das veränderte Verhalten verantwortlich ist, wenn er weiß, wie er damit umzugehen hat. Je nach MMU Typ scheint es auf ST/STE möglich zu sein, etwa 13 bzw. 29 Zeilen mehr oberhalb des normalen vom ST/STE dargestellten Bildausschnitt, darzustellen. Nun, dazu muß man an dieser Stelle wieder eine 50/60 Hz Umschaltung, eine kleine Pause und eine 60/50 Hz Umschaltung vornehmen. Da an dieser Stelle noch vom GLUE noch kein DE (Display Enable) Signal kommt, ist es nicht möglich, den Event Count Mode des Timer B zum Auffinden dieser Stelle zu nutzen. Es gibt meines Wissens zwei Möglichkeiten, die Zeile (echte Bilschirmzeile) zu finden, an der man umschalten muß. Zum einen kann man im VBL alle Interrupts außer dem HBL ($68 wird normalerweise nicht benutzt und hat eine sehr niedrige Priorität) sperren und die HBL's zählen. Nach etwa 33 HBL's hält man mit STOP #$2300 den Prozessor an, dieser wartet dann bis zum nächsten HBL. Dann noch etwa 86 nop's warten, auf 60 Hz schalten, wiederum ca. 14-17 nops und wieder auf 50 Hz schalten. Eigentlich müßte der obere Rand sich dann öffnen. Natürlich ist auch die Kombination vom geöffneten oberen und unteren Rand möglich. Die zweite Methode setzt einen Timer in der letzten Zeile, und wartet bis die richtige Stelle ereicht ist; natürlich müssen alle anderen Interrupts (die stören könnten) währendessen ausgeschaltet sein.

1.4.1.3 Das Öffnen aller Ränder

Ich will es kurz machen, denn die Sache ist zwar hübsch anzuschauen - aber eine "elendige Taktzyklenzählerei". Man sucht sich nach der oben genannten Methode (Öffnen des oberen Bildsschirmrands) den Anfang des oberen, geöffneten Bildschirmrands, öffnet ihn und syncronisiert den Prozessor mit dem Aufbau einer

Zeile, setzt an den richtigen Stellen in den Zeilen eine 50/60 Hz bzw. 50/71 Hz Umschaltung vor, und öffnet damit den linken und rechten Rand.

Es ist fast unmöglich, zum Öffnen aller Ränder zusätzlich bewegte Grafik und/oder digitalisierten (YAMAHA) Sound zu bringen. Das einige Demoprogrammierer das doch geschafft haben, liegt an der Möglichkeit, statt den NOP's andere Prozessorbefehle zur Bildsynchronisation zu nutzen und somit den Programmteil für Grafik und Sound, in die Routine zum Öffnen der Ränder, quasi "hineinzuweben". Ein mächtig kompliziertes Unterfangen !

1.4.2 Der "sogenannte" Syncscroller

Eigentlich wird dabei nur das fehlende Low Byte der Video Basis Register im ST simuliert. Bei STE/TT existiert so ein Register, eine Simulation ergibt somit bei diesen Geräten nur wenig Sinn.

Wenn man die Techniken zum Öffnen der Bildschirmränder verstanden hat, ist die Sache mit dem Syncscrolling relativ einfach zu realisieren (man muß eben darauf kommen !). Im Prinzip basiert der Syncscroller auf der Eigenschaft des Bildschirmspeichers größer zu werden, wenn man mehr Pixel in einer Zeile darstellt. Das klingt eigentlich recht banal, so muß man nur die Ränder von einigen Zeilen öffnen (mal den linken und mal den rechten Rand) und schon verschiebt sich der Bildschirmspeicher. Am unauffälligsten ist es, die geöffneten Zeilen in den oberen Rand (den muß man natürlich vorher öffnen) zu plazieren, dann schaltet man im VBL alle Farbregister dunkel und setzt sie erst nach der Öffnung der Ränder wieder auf sichtbare Farbwerte und schon kann keiner sehen, daß irgendwelche Manipulationen im oberen Rand vorgenommen worden sind.

1.5 Die VIDEO - Hardwareregister im Falcon030

... oder was eigentlich keiner wissen oder benutzen darf

Vielen Dank an Torsten Lang für seine hilfreichen Tips zu diesem Thema!

Achtung ! **** UNDOKUMENTIERT ****

Benutzung strengstens verboten !!!

Ich übernehme keine Gewähr auf Richtigkeit der hier angegebenen Daten. Die hier angegebenen Register habe ich durch Versuchen und teilw. durch Disassemblieren des Betriebssystems gefunden. ATARI hat keines dieser Register dokumentiert (wahrscheinlich wird dies auch nicht passieren) und hat die neuen Register im F030 in manchen Fällen so ungünstig positioniert, daß es in vielen Fällen zu Problemen mit vorhandener Software, vorrangig Spielen/Demos, kommt, die trotz der Möglichkeit den F030, auf ein 8 MHz Timing zu schalten, ihren Dienst verweigern. Schade, da hätte mehr Rücksicht auf vorhandene Applikationen, mit relativ geringem Aufwand betrieben werden können. Trotz alledem ist der F030 eine faszinierende Maschine, leider hat es ATARI verpaßt, der doch recht ausgereiften VIDEO - Hardware des F030 ebenso mächtige Betriebssystem-Funktionen zur Verfügung zu stellen; aus diesem Grund und für die Programmierer systemnaher Software habe ich mich entschlossen, die Video-Hardware-Register des F030 (vorläufig) zu dokumentieren.

Wer sich die neuen Register des F030 anschaut, wird bei der Leistungsfähigkeit an Video-Prozessoren wie den Tseng ET4000 oder den ARABELLA Video Chip des Archimedes erinnert.

Man verzeihe mir bitte die Ungenauigkeiten, aber es ist nicht immer einfach, die Funktion eines Registers durch seine Wirkung zu erkennen.

Falcon Shift-Mode-Register

$ffff8266 W %XXXXXXXT XVHPXBBB

T = True Color / 65536 Farben Bit
V = externe vertikale Synchronisation
H = externe horizontale Synchronisation
P = 256 Farben

BBB = Bank zu 16 Farben, die in 16er Schritten aus der 256 Farbpalette "CLUT" gewählt werden kann.

Wenn weder 'T' noch 'P' gesetzt ist, befindet sich der Rechner im 16 Farb Modus,

Das "256 Farben" Bit funktioniert nur, wenn das "True Color" Bit gelöscht ist.

In diesen Modi gilt die Falcon 18 Bit-Palette

Horizontale Position

$ffff828c W %nnnnnnnn nnnnIPPP RW h_pos

I = Interlace an
PPP = Verschieben des Bildbereichs von links nach rechts

Wirkung:
Damit läßt sich der dargestellte Bildbereich nach links oder rechts schieben, so weit der Monitor es schafft, das Bild zu synchronisieren. Wenn das 'I' Bit gesetzt ist, ergibt sich die Interlace Darstellung.

Umstellen auf entsprechenden Monitor Typ

PAL/NTSC/VGA

$ffff82c0 W %nnnnnnnP nMnnnVnn RW mon_typ

Wenn Bit gesetzt

P = PAL flag (z.B. Deutsches Fernsehen)
N = NTSC flag(Fernsehnorm in den USA)
M = Monocrome flag(z.B. SM 124/144 etc)

Beim gleichzeitigen Setzen mehrerer Bits entstehen interessante Effekte!

Zeilenwiederholfrequenz

$ffff8282 W %XXXXXXXn nnnnnnnn RW h_cycle

Horizontal Blank rechts (nicht in ST hoch 71Hz)

$ffff8284 W %XXXXXXXn nnnnnnnn RW sr_hblank

Mit diesem Register läßt sich einstellen, wann in der Zeile (ab der Mitte bis an den rechten Rand) der Videochip dunkel schaltet.

Horizontal Blank links (nicht in ST hoch 71Hz)

$ffff8286 W %XXXXXXXn nnnnnnnn RW sl_hblank

Wie Register zuvor, nur von der Mitte zum linken Rand.

Linker Rand

$ffff8288 W %XXXXXXXn nnnnnnnn RW r_border

Rechter Rand

$ffff828A W %XXXXXXn nnnnnnnn RW l_border

Horizontal Sync Start

$ffff828c W %nnnnnnnn nnnnnnnn RW h_sync_start

Bildwiederholfrequenz

$ffff82a2 W %XXXXnnnn nnnnnnnl RW v_cycle

l=Interlace, wenn gesetzt, Interlace aus

Vertikal Blank unten

$ffff82a4 W %XXXXXXXn nnnnnnnn RW h_dis_start

Vertikal Blank oben

$ffff82a6 W %XXXXXXXn nnnnnnnn RW h_dis_end

Oberer Rand

$ffff82a8 W %XXXXXnnn nnnnnnnn RW v_dis_start

Beginn der nutzbaren Bildschirmfläche

Unterer Rand

$ffff82aa W %XXXXXnnn nnnnnnnn RW v_dis_end

Ende der nutzbaren Bildschirmfläche

Vertical Sync Start

$ffff82ad W %XXXXXXnnn nnnnnnnn RW v_sync_start

Sync Mode Falcon

$ffff82c0 W %XXXXXXXP nnnnnVnn sync_fal

Wenn gesetzt : P = Pal/NTSC
V = VGA

Pixel Höhe/Breite

$ffff82c2 W %XXXXXXXX XXXX EBZH RW resolution

H = Pixelhöhe, wenn gesetzt, doppelte Höhe
Z = Zeilensprung, wenn gesetzt, wird je eine Zeile beim Auslesen des Bildschirmspeichers übersprungen
B = Breite , wenn gesetzt, "320" Pixel pro Zeile sonst "640"
E = Externe Synchronisation oder Pixelclock ???

Farbpalette - 18 Bit = 262144 Farbtöne

$FFFF9800 L %rrrrrrXX ggggggXX XXXXXXXX bbbbbbXX RW Fal_col0

bis

$FFFF9BF8 L %rrrrrrXX ggggggXX XXXXXXXX bbbbbbXX RW Fal_col255

Achtung, auf diese Register darf nur lang/wortweise zugegriffen-werden, ein byteweiser Zugriff wirkt sich sowohl auf das niederwertige als auch auf das höherwertige Byte des jeweiligen Farbregisters aus.

B.2. Speicher schaufeln mit dem Blitter

Ende 1987 wurde die ersten Blitter in die MEGA ST's serienmäßig eingebaut. Ende 1989 wurden die ersten 1040 STE's vekauft, in dessen Grafikchips wurde der Blitter gleich integriert. Ende 1990 konnte man (endlich) den TT030 kaufen, leider enthält dieser keinen Blitter !

Seit Anfang 1993 kann man den FALCON030 kaufen, in dessen Grafikchip wurde der Blitter auch gleich integriert und mit doppelter Taktfrequenz (16 Mhz) betrieben, d.h. er ist etwa doppelt so schnell wie der Standard-Blitter in den ST(E) Computern.

Nun, das Fehlen eines Blitters fällt im TT eigentlich nicht ins Gewicht, jedoch bedeutet das für den Programmierer, daß er nicht davon ausgehen kann, daß die ATARI Computer ohne Blitter irgendwann aussterben würden. Was wiederum bedeutet, daß man mindestens zwei Bitblit- oder Sprite-Routinen schreiben muß, wenn man die Geschwindigkeit der Geräte voll ausnutzen möchte. Zum einen, eine die den Blitter nutzt und zum anderen natürlich eine, die das Ganze mit dem Prozessor erledigt (wer es richtig machen will, schreibt noch eine, die den 68030 voll ausnutzt). Ich kenne (leider) nur ein Spiel, das auch den Blitter nutzt, es ist 'Wings of Death' von Thalion, das allerdings hartnäckig den Dienst auf dem TT verweigert.

Programmierer, die auf dem ST/STE/TT/Falcon ... professionelle Anwender-Software schreiben, werden sich wohl "die Haare raufen", wenn sie schon wieder was von direkter Programmierung der Hardware hören, es ist jedoch (fast) unmöglich, gute GEM konforme Spiele mit flüssiger Animation zu schreiben, weil weder AES noch VDI Möglichkeiten bietet, mehrfarbige Sprites oder sonst irgend etwas mit dem Bildaufbau des Monitors zu synchronisieren. Es bleibt uns also nichts anderes übrig, als in den "sauren Apfel" zu beißen und den Blitter selbst zu programmieren. "Immer ?" - "Nein, nicht immer - aber ..."

2.1 Die Hardwareregister des Blitters

Der Blitter ist fast "nur" die hardwaremäßige Realisierung des Line A - Bitblit Befehls bzw. Copy Raster, Opaque (VDI 109) des VDI (der ja auf Bitblit zurückgreift). Allerdings untestützt der Blitter auch das Füllen von Flächen mit Mustern, da er einen speziellen Speicher für Füllmuster, das Halftone-Ram, besitzt. Wie in anderen Kapiteln erwähnt, ist der ATARI nicht speziell als Spielkonsole entwickelt worden und so wurde auch der Blitter eher als Unterstützung des Betriebssystems konzipiert. Aber es gibt eben nichts, was man nicht in irgendeiner Weise zum Spielen "mißbrauchen" könnte.

Spötter behaupten, der Blitter wäre der Versuch Atari's gewesen, das durch die Unfähigkeit seiner Programmierer zu langsam geratenen VDI durch Verbesserung der Hardware zu beschleunigen.

2.1.1 Register zur Definition des Quellbitfelds

Halftone-Ram:

$ff 8A00 bis $FF 8A1E (16 Wörter)

Den Inhalt des Halftone-Ram kann man als Quelle als auch zur Verknüpfung mit den Quelldaten nutzen.

Source X Increment:

$ff 8A20 W %nnnnnnnn nnnnnnnX RW Src_Xinc

Gibt an, wieviel Bytes der Blitter zwischen den zu lesenden Wörtern der Quelldaten überspringen soll. Nimmt vorzeichenbehaftete Integer-Werte an und wertet sie gemäß ihres Vorzeichens aus.

Source Y increment:

$FF 8A22 W %nnnnnnnn nnnnnnnX RW Src_Yinc

Gibt den Offset in Bytes zwischen dem letzten Wort einer Zeile zum ersten der nächsten Zeile an. Nimmt vorzeichenbehaftete Integer-Werte an und wertet sie gemäß ihres Vorzeichens aus.

Source-Adress-Register:

$FF 8A24 L %XXXXXXXX nnnnnnnn nnnnnnnn nnnnnnnX RW Src_Addr

Hier wird die Anfangsadresse der Quelle eingetragen.

Masken - Register:

Dienen zum Ausmaskieren der Bits in den Wörtern des Zielbitfeldes.

Die in diesen Registern gesetzten (!) Bits werden im Zielbitfeld ausmaskiert.

$FF 8A28 W %nnnnnnnn nnnnnnnn RW Endmask1

Nur für das erste Wort einer Zeile.

$FF 8A2A W %nnnnnnnn nnnnnnnn RW Endmask2

Für alle Wörter zwischen erstem und letztem Wort einer Zeile.

$FF 8A2C W %nnnnnnnn nnnnnnnn RW Endmask3

Nur für das letzte Wort einer Zeile.

2.1.2 Register zur Definition des Zielbitfeldes

Destination X increment:

$FF 8A2E W %nnnnnnnn nnnnnnnx RW Dst_Xinc

Funktion wie bei Source X increment !

Destination Y increment:

$FF 8A30 W %nnnnnnnn nnnnnnnX RW Dst_Yinc

Funktion wie bei Source Y increment !

Destination Adress:

$FF 8A32 L %XXXXXXXX nnnnnnnn nnnnnnnn nnnnnnnX RW Dst_Addr

Startadresse des Zielbitfeldes

X-Count-Register:

$FF 8A36 W %nnnnnnnn nnnnnnnn RW X_Count

Gibt die Anzahl der Worte in einer Zeile an. Bei Abarbeitung einer Kopieroperation wird das Register mit jedem bearbeiteten Wort heruntergezählt. Wenn das Register 0 ist, wird es wieder auf seinen Startwert gesetzt.

Y-Count-Register:

$FF 8A38 W %nnnnnnnn nnnnnnnn RW Y_Count

Gibt Anzahl der zu bearbeitenden Zeilen an.
Bei Abarbeitung einer Kopieroperation wird das Register mit nach Bearbeiten einer Zeile heruntergezählt. Wenn das Register Null ist, wird es wieder auf seinen Startwert gesetzt.

2.1.4 Operations Register des Blitters

Halftone Operation Register:

$FF 8A3A B %XXXXXnnn   RW   HOP

Wird gesetzt je nach Verwendung des Halftone-RAM !

Inhalt	Bedeutung
0	Alle Bits
1	Nur Halftone Daten
2	Nur Source Daten
3	Source AND Halftone

Operation Register:

$FF 8A3C B %XXXXnnnn   RW   OP

Wie bei RC_COPY oder PUT und GET in GFA - BASIC wird hier die Art der Verknüpfung festgelegt.

Inhalt	Bedeutung
0	Alle Bits werden gelöscht
1	Quelle AND Ziel
2	Quelle AND NOT(Ziel)
3	nur Quelle
4	NOT(Quelle) AND Ziel
5	Ziel
6	Quelle XOR Ziel
7	Quelle OR Ziel
8	NOT(Quelle) AND NOT(Ziel)
9	NOT(Quelle) XOR Ziel
A	NOT(Ziel)
B	Quelle OR NOT(Ziel)
C	NOT(Quelle)
D	NOT(Quelle) OR Ziel
E	NOT(Quelle) OR NOT(Ziel)
F	Alle BITS werden gesetzt

Line Number:

$FF 8A3C B %nnnXnnnn RW Line_Num

Bit	Wirkung
0-3	Line Number Gibt an, welches Wort des Halftone-RAM verwendet wird. Wird abhängig vom 'Dst_Yinc' Register (je nach vor Vorzeichen) erhöht bzw. erniedrigt.
5	Smudge Wenn das Bit gesetzt wird, betimmt der Inhalt der untersten 4 Bit des Quellbitfelds, welches Wort des Halftone-RAMs verwendet wird.
6	HOG Wenn dieses Bit gesetzt ist, hat der Blitter Vorrang bei Buszugriffen. Wenn es gelöscht ist, wechseln sich Blitter und Prozessor alle 64 Buszyklen ab.
7	BUSY Startet den Blitter

Das Skew Register:

$FF 8a3D B %nnxxnnnn RW Skew

Bit	Wirkung
0-3	Skew Gibt an, um wieviel Stellen die Datenwörter, die vom Quellbitfeld stammen, nach rechts geshiftet werden, wobei die überzähligen Bits ins folgende Datenwort geschoben werden.
6	NFSR (**N**o **F**inal **S**ource **R**ead) Ist dieses Bit gesetzt, wird das letzte Datenwort einer Zeile der Quelle ignoriert.
7	FXSR (**F**orce **EX**tra **S**ource **R**ead) Ist dieses Bit gesetzt, wird noch ein zusätzliches Wort am Anfang jeder Zeile gelesen.

2.2 Was man bei der Blitter-Programmierung beachten sollte !

Prinzipiell funktioniert das Bewegen von Blöcken von einem Speicherbereich in den anderen mit dem Blitter genauso einfach wie mit einer selbstgeschriebenen Routine. Allerdings nimmt der Blitter dem Programmierer den kompliziertesten Teil der Arbeit (das Programmieren einer Routine !) ab, so daß man eigentlich nur Startadresse des Blocks, Zieladresse des Blocks, Zeilenbreite usw. eingeben muß, die restlichen Register mit Default-Einstellungen belegen muß und schon macht der Blitter (fast) alles von allein. Dazu befindet sich ein Beispiel-Listing auf der Diskette !

2.2.1 Welche Register ändern sich nach dem Start ?

Wichtig ist zu wissen, daß man beim erneuten Starten des Blitters nicht alle Register neu beschreiben muß. Nur die Register mit den Startadressen der Quelle und die Menge der zu kopierenden Zeilen werden effektiv vom Blitter verändert, alle anderen Blitterregister (außer dem Busy-Bit zum Starten des Bitters im Line_Num Register) behalten ihre alten Werte. In vielen Fällen kann man somit durch Beschreiben von 12 Bytes im Blitterregister ganze Bildschirmseiten kopieren. Natürlich darf man nach Benutzen von LINE A oder VDI, falls man nicht mit Blitmode dem VDI/LINE A die Benutzung des Blitters verboten hat, alle Blitterregister wieder initialisieren.

2.2.2 Nutzung des Blitter in unterschiedlichen Auflösungen

Muß man, durch die etwas ungünstige Lage der Bitplanes, in den Auflösungen bis 8 Bit Tiefe den Blitter für jede Bitplane einzeln bemühen, kann man in den 16 Bit "Echtfarbauflösungen" des Falcon030 mit einem einzigen Start des Blitters einen Rasterkopierbefehl erledigen, da in dieser Auflösung ein Bildpunkt einem Datenwort des Blitters entspricht.

Einstellung für X - Increment je nach Auflösung

mono/duocrome	1
4 Farben	4
16 Farben	8
256 Farben	16
"TRUE" Color	1

Bei den paletteorientierten Auflösungen muß man beim Kopieren eines Bitblocks pro Plane eine Blitteroperation durchführen lassen.

Im Falcon "TRUE" Color Mode lassen sich durch direkte Programmierung des Blitter interessante Effekte erzielen. So läßt sich, einfach durch Setzen des X - Increments der Quelle auf zwei, der Bildschirm horizontal halbieren. Auch kann man durch zusätzliche Programmierung des X - Increments des Ziel und entsprechenden Kopieroperationen den Bildschirm in Y Richtung leicht um das X - fache dehnen oder stauchen. Natürlich lassen sich in allen Auflösungen mit Hilfe des Blitters solche Effekte in Y Richtung erzeugen.

2.2.3 Die Betriebsmodi des Blitters

Wie schon erwähnt, läßt sich mittels des HOG Bits im 'Line_num' Register die Betriebsart des Blitters festlegen. Bei gelöschtem Bit teilen sich Prozessor und Blitter den Zugriff auf den Bus im Wechsel von 64 Zyklen, wobei jeweils Blitter oder Prozessor, während der andere dran ist, nicht mehr auf den Bus zugreifen können. Da der Blitter am IO3 Port des MFP angeschlossen ist, kann man einen Interrupt auslösen lassen, wenn der Blitter seine Operation beendet hat.

Wenn das HOG Bit gesetzt ist, bekommt die CPU nur noch den Bus, wenn der Blitter ihn zwischendurch frei gibt. Da der Blitter jetzt bei jedem Taktzyklus zum Zug kommt, und nicht nur alle 64 Zyklen mal 64 Zyklen, ist er in diesem Modus doppelt so schnell wie bei gelöschtem HOG Bit. Der Nachteil ist jedoch, daß während

der Blitter läuft, der Prozessor keine Interrupts mehr durchführen kann. Wenn also der Blitter ohne Unterbrechung läuft, reicht die Zeit möglicherweise nicht mehr für den Maus/Keyboard Interrupt, d.h. Daten der Pakete vom IKBD (Intelligent **K**ey**B**oar**D** Processor) nicht abgeholt werden können und somit unvollständig ans Betriebssystem geliefert werden, was den Mauspfeil recht seltsam reagieren lassen kann !

2.3 Betriebssystem-Unterstützung des Blitters

Ab TOS 1.02 wird vom Betriebssytem beim Sytemstart das Vorhandensein eines Blitters überprüft. Das TOS testet die Anwesenheit des BLitters durch den Versuch, dessen Register zu beschreiben; ein ST ohne Blitter reagiert auf einen solchen Versuch mit einem Busfehler, die 68000er Familie springt bei einem Busfehler an eine Stelle, die, fall das Vector Basis Register Null ist, an der Adresse $08 steht. Das Betriebssystem schreibt vor dem Test an $08 die Adresse einer Routine, die dann ein internes Flag setzt, was soviel bedeutet wie: kein Blitter vorhanden. Nachfragen kann man das ganze via XBIOS 64.

Nomenklatur:

WORD = 16 Bit
LONG = 32 Bit
VOID = Dummy, d.h. keine Rückgabe
* = Zeiger auf

XBIOS 64

WORD Blitmode(WORD mode)

Mit diesem Aufruf kann man dem Betriebssystem die Benutzung des Blitters untersagen (nicht beim Falcon030), so daß das VDI(ROM -Treiber)/LINE A nur die CPU zum Realisieren der Grafikfunktionen gebrauchen dürfen.

Aufruf in Assembler:

```
move.w    #mode,-(sp)
move.w    #64,-(sp)
trap      #14
addq.l    #4,sp
```

mode, -1 gibt eingestellten Wert zurück
Bit 0 gesetzt , Blitter an
Bit 0 gelöscht, Blitter aus

gibt zurück:

Bit 0 = 0 Blitter wird nicht benutzt
= 1 Blitter wird benutzt
Bit 1 = 0 kein Blitter vorhanden
= 1 Blitter vorhanden

Bit 15 immer 0

alle restlichen Bits sind reserviert !

B.3. Der MFP , so wie ihn ein Spieleprogrammierer benutzt

Was des einen Copper (AMIGA CHIP) ist des anderen MFP !

Wer sich einige der neueren Spiele für den ST angesehen hat, sieht Farben, nichts als Farben. Wie war das noch mal (?) nur vier Bitplanes (!) das sind doch "nur" sechzehn Farben ??? Ja, im Prinzip schon (!) aber man kann die Farbregister des Shifters beschreiben und dieser reagiert sofort auf den neuen Inhalt.

Auch gibt es einige Adventures, die bis zu einem bestimmten Punkt eine LORES (320*200*4) 16 Farb-Darstellung zeigen aber an der Stelle wo man die Komandos eingibt, diese in der mittleren Auflösung (640*200*2) eingeben darf. Das liegt wiederum darin begründet, daß der Inhalt eines Hardwareregisters sofort in den Shifter übernommen wird. In diesem Fall ist es das SHIFT MODE Register.

Nun, wenn diese Register sofort übernommen werden, dann muß man sie immer exakt zum richtigen Zeitpunkt beschreiben, um synchron zum Bildaufbau die Farb oder Auflösungsumschaltung vorzunehmen, denn wenn man das nicht schaft, werden die Bilder nicht mehr ganz so schön wie man sich das eigentlich vorgestellt hatte.

Viele Spieleprogrammierer versuchen, eine Vielzahl von Farben auf dem Bildschirm zu erreichen, indem sie den Bildaufbau mit dem mit den Ausführungszeiten der Befehle synchronisieren (siehe dazu auch Kapitel B.1.4). Jedoch bekommt man damit unwillkürlich Probleme, wenn der Rechner mit einen schnelleren Prozessor ausgerüstet ist (wie z.B. beim TT oder Falcon dem Fall), da das Programm, aufgrund der höheren Ausführungsgeschwindigkeit, das Bild schneller im Speicher aufbaut und die Farben umschaltet als der Monitor für den Bildaufbau auf dem Bildschirm benötigt. Probleme gab's mit einigen Spielen auf dem TT, die dann entweder beim Bildaufbau flackerten oder unbedingt auf 50Hz schalten wollten, weil ihre Routine zum Bildaufbau auf diese Frequenz syncronisiert worden ist. Nun, ein Chip, der - egal wie hoch der Prozessor getaktet ist - immer zur richtigen Zeit umschaltet, ist der MFP mit seinen Timern.

3.1 Der MFP (Multi Function Peripheral)

Der MFP 68901 ist, wie der Name schon sagt, ein Peripherie-Baustein, der sehr universell einsetzbar ist. Auf die technischen Einzelheiten zu diesem Chip wird schon in vielen anderen Büchern eingegangen, daher möchte ich nur das für den Spieleprogrammierer Wichtige erwähnen.

Jeder ST besitzt einen MFP, der TT sogar zwei. Jeder MFP, sowohl der im ST/STE/Falcon als auch die zwei im TT, werden mit 2,4756MHz getaktet. Alle Interrupts der Timer werden durch Vorteiler aus diesem Systemtakt erzeugt.

Der MFP besitzt verschiedene Ein/Ausgänge, die je nach Einstellung bei einer Änderung der angelegten Spannung, von High (5V) auf LOW (0V) einen Interrupt auslösen können. Wichtig für den Spieleprogrammierer ist dabei der Timer B, der an dem Display Enable Signal des Video-Sub-Systems anliegt, was wiederum nicht anderes bedeutet, daß es bei entsprechender Einstellung möglich ist, alle n<256 Zeilen einen Interrupt auszulösen um z.B. die Farbregister zu ändern (dazu später mehr). Bei STE/TT/Falcon liegt am Timer A Eingang (unter anderem) ein Signal vom DMA-Sounchip an, der beim letzten Füllen seines Abspielpuffers einen Interrupt auslöst. Auch erzeugt der MFP einen Interrupt, wenn der Blitter seine Arbeit beendet hat. Diese Verschaltung des MFP mit anderen Bausteinen der Video/Audio Hardware machen diesen Bustein für uns Spieleprogrammierer höchst interessant. Interessant ist natürlich auch, daß der MFP einen Interrupt erzeugt, damit die CPU die Maus-, Joystick- und Keyboard-Daten von den ACIAS (Übertragungsbausteine), die mit dem IKBD (Prozessor für die Verwaltung von Keyboard-, Joystick- und Maus-Daten) abholt.

Im Prinzip wird also fast alles verwaltet, was wir so brauchen. Daher ist es wichtig zu wissen, wie man ihn programmiert.

3.2 Die Programmierung der MFPs

3.2.1 Die Hardwareregister des ST - MFP

Achtung, ich nenne den MFP, der in allen ATARI Rechnern (STE/TT/F030) enthalten ist, ST - MFP.

Gpip Data Register

```
$fffa01 B  %nnnnnnnn  R        gpip
            | | | | | | | |      Bit
            | | | | | | | |___   0  Centr. Port, Drucker beschäftigt
            | | | | | | |_____   1  RS 232, Data Carrier Detect
            | | | | | |_______   2  RS 232, Clear to send
            | | | | |_________   3  GPU Done, Blitter fertig
            | | | |                   CLKDIR (nur im TT)
            | | | |___________   4  IRQ, MIDI - Tastatur ACIA
            | | |_____________   5  Meldung von ASCI/FDC
            | |_______________   6  RS 232 Ring Indikator
            |_________________   7  Monochrome Monitor Detect
                                    beim (STE/TT) XOR mit
                                    Frame-End Signal vom
                                    DMA Sound Subsystem
```

Liefert die logischen Pegel der Anschlüsse. Nun zu den für uns wichtigen Bits !

Bit 3, GPU Done

Wenn dieses Bit gelöscht ist, hat der Blitter seine Arbeit beendet. Siehe dazu auch das Kapitel B.2 zur Blitterprogrammierung.

Bit 4, MIDI/Tastatur ACIA

Je nach Modus liefert der IKBD (Tastaturprozessor) Maus, Joystick und Tastatur Daten, die von der CPU per Interrupt in den Registern der ACIA (Asyncron-Communications-Interface-Adapter) abgeholt werden müssen.

Bit 7, Monocrome Detect, bei STE/F030/TT XOR mit

Frame-End-Signal

Im VBL fragt das Betriebssystem das Monochrome Detect Bit ab, bei gelöschtem Bit ist der Rechner an einen monochromen Monitor, bei gesetztem Bit an ein Farbmonitor angeschlossen. Wenn die eingestellte Auflösung nicht mit dem angeschlossenen Monitor übereinstimmt, wird im VBL ein Warmstart (Reset ohne Löschen des Speichers) ausgelöst und damit die Auflösung geändert. Bei einem ATARI mit 8 Bit DMA Sound-Subsystem wird das Monochrome Monitor Detect Signal noch mit dem DMA Sounchip XORed exklusiv geodert, siehe dazu auch Kapitel B.3 über Sounderzeugung.

Das Active Edge Register

$fffa03 B %nnnnnnnn RW aer

Belegung der Bits siehe 'gpip'

Sinn dieses Registers ist es festzulegen, wann ein Interrupt ausgelöst wird. Das hängt mit der Interruptquelle, die am MFP anliegen zusammen. Wenn am Eingang einer Quelle plötzlich 0V (low) statt 5V (high) anliegen, wird nur dann ein Interrupt ausgelöst, wenn das entsprechende Bit (siehe gpip) gesetzt ist. Bei gelöschtem Bit wird ein Interrupt bei einer Änderung von high auf low ausgelöst.

Das Data Direction Register

$fffa05 B %nnnnnnnn RW ddr

Belegung der Bits siehe 'gpip'

Man kann mit diesem Register angeben, ob die I/O's des MFP als Eingänge oder Ausgänge verwendet werden, da die bisher verwendeten I/O' Leitungen alle als Eingänge verschaltet sind, sind die Bits in diesen Registern alle gelöscht.

Das Interrupt-Enable-Register A

```
$fffa07 B % nnnnnnnn RW     iera
            | | | | | | | |  Bit
            | | | | | | | |__ 0  Timer B (Display enable Signal)
            | | | | | | |____ 1  XMIT Fehler (RS 232)
            | | | | | |______ 2  XMIT Puffer leer (RS 232)
            | | | | |________ 3  RCV  Fehler (RS 232)
            | | | |__________ 4  RCV  Puffer voll (RS232)
            | | |____________ 5  Timer A
            | |                  Timer A (Frame end bei
            | |                  STE/TT/Falcon 8 Bit DMA Sound)
            | |______________ 6   Port I6 (RS 232 Ring Indikator)
            |________________ 7  Port I7 (Monochrome Detect)
```

Mit diesem Register kann man das Erzeugen eines Interrupts verhindern, indem man das entsprechende Bit aus dem Register löscht.

Die wichtigsten Bits:

Bit 0, Timer B (Display enable Signal)
Durch das Einschalten dieses Interrupts ist es möglich, in einer beliebigen Bildschirmzeile (0 bis 255) einen Interupt auslösen zu lassen. Mit entsprechender Programmierung ist es damit möglich, den Bildschirm in mehrere Zonen mit unterschiedlichen Farben einzuteilen (sogenannte Rasterinterrupts) oder änllich wie schon bei den 8 Bit-Atari's von einer zur anderen Zeile die Auflösung zu wechseln (bei den 8 Bit-Geräten bekannt unter dem Namen "Display Listen Interrupt"). Auch Splitscreens (bei STE/TT/Falcon) und bei STE/Falcon sogar unabhängiges Fine-Scrolling der Splitscreens wird durch geschickte Programmierung möglich. Nicht zu vergessen, ohne diesen Interrupt wäre das Öffnen des unteren Bildschirmrandes bei ST/STE "unmöglich". Allerdings kann man den Timer B auch "nur" als Timer benutzen, der nach einer gewissen Zeit einen Interrupt auslöst.

Bit 5, Timer A
Natürlich ist auch der Timer A ein Timer, der nach einer gewissen Zeit einen Interrupt auslösen kann. Allerdings ist es auch möglich, sofern es sich um einen STE/TT/Falcon handelt, den Timer A im Event-Count-Modus DMA-Sound Frame End's (siehe Kapitel 4) zählen und gegebenenfalls einen Interrupt auslösen zu lassen.

Bit 7, Port I7 (Monochrome Detect)
Wird im Atari nicht benutzt, ob ein MONOCHROME Monitor angeschlossen ist, wird vom Betriebssytem während des VBL's am 'GPIP' Bit 7 getestet. Auch dieser Interrupt kann beim STE/TT/Falcon030 bei Frame End ausgelöst werden.

Das Interrupt Enable Register B

```
$fffa09 B %  nnnnnnnn  RW    ierb
             | | | | | | | |   Bit
             | | | | | | | |__ 0  Port I0 (Busy Centronics)
             | | | | | | |____ 1  Port I1 (RS232 Data Carrier Detect)
             | | | | | |______ 2  Port I2 (RS232 Clear To Send)
             | | | | |________ 3  Port I3 (GPU Done, nur mit Blitter)
             | | | |__________ 4  Timer D (Baudrate RS232)
             | | |____________ 5  Timer C (200 Hz System (AES etc.))
             | |______________ 6  Port I4 (IRQ IKBD/Maus/MIDI)
             |________________ 7  Port I5 (IRQ Floppy Hardd. DMA)
```

Funktion wie Interrupt-Enable-Register A !

Die wichtigsten Bits:

Bit 3, I3 (GPU done)

Erzeugt einen Interrupt, wenn der Blitter seine Arbeit beendet hat, siehe dazu auch Blitterprogrammierung.

Bit 5, Timer C (200Hz Timer)

Dieser Interrupt bietet sich an, auf eigene Soundroutinen zum Abspielen von Musik verwendet zu werden. Dabei sollte man Vektor 256 benutzen, der 50 mal pro Sekunde, also bei jedem vierten Aufruf von Timer C, von der Timer C Routine aufgerufen wird. Diesen Vektor kann man mit 'Setexec' also BIOS 5 auf eine eigene Routine umbiegen.

Bit 6, Port I4 (Joystick, Keyboard, MIDI, Mouse)

Dieser Interrupt wird immer dann ausgelöst, wenn ein Paket vom IKBD abgeholt werden soll, wobèi das dann Maus, Keyboard oder Joystickdaten sein können. Außerdem wird der Interrupt ausgelöst, wenn am MIDI Port Daten anliegen. Die Daten müssen dann von der jeweiligen ACIA (**A**syncrone - **C**ommunications - **I**nterface - **A**dapter) abgeholt werden. Von welcher ACIA die Daten abzuholen sind, kann die Interruproutine durch Prüfen der ACIA Statusregister ermitteln.

Das Interrupt Pending Register A

$fffa0B B %nnnnnnnn RW ipra

Belegung siehe Interrupt Enable Register A

Entsprechende Bits werden beim Auftreten eines Interrupts gesetzt und nach dem Erzeugen des Vektors - oder wenn der Prozessor den Interrupt anerkannt hat - wieder gelöscht. Hat man zuvor selbst das entsprechende Bit gelöscht (Dazu muß ein Byte in das Register geschrieben werden, in dem alle Bits gesetzt sind, die nicht gelöscht werden sollen) wird der Interrupt nicht ausgelöst.

Das Interrupt Pending Register B

$fffa0d B %nnnnnnnn RW iprb

Belegung siehe Interrupt Enable Register B
Funktion siehe Interrupt-Pending-Register B

Das Interrupt In Service Register A

$fffa0f B %nnnnnnnn RW iera

Belegung siehe Interrupt Enable Register A

Wie schon der Name schon sagt, zeigt dieses Register an, wann ein Interrupt abgearbeitet wird. Das entsprechende Bit in diesem Byte wird dann gesetzt, wenn das Bit im Pending Register gelöscht wird. Im Automatic End of Interrupt (AEOI) wird das Bit aber dann auch gleich wieder gelöscht und für den MFP ist dieser Interrupt beendet, es können also neue Interrupts oder nochmal die gleiche Interruptanforderung auftreten, diese werden ausgeführt, wenn in der gerade laufenden Interruptroutine der Interrupt Level im Statusregister heruntergesezt wurde. Im Software End Of Interrupt

(SEOI) kann kein MFP-Interrupt mit niedrigerem Interruptlevel als dem gerade laufenden ausgeführt werden, solange das In Service Bit gesetzt ist. Für Interrupts mit niedrigerem Level wird aber das Pending Bit gesetzt, was wiederum das Erzeugen eines Vektors nach dem Löschen des Interrupt in Service Bits nach sich zieht.

Das Interrupt in Service Register B

$fffa11 B %nnnnnnnn RW ierb

Belegung siehe Interrupt Enable Register B
Funktion siehe Interrupt in Service Register A

Das Interrupt Mask Register A

$fffa13 B %nnnnnnnn RW imra

Belegung siehe Interrupt Enable Register A

Mit diesem Register ist es möglich, angeforderte Interrupts auszumaskieren. Was so viel bedeutet wie, die Interrupts werden dem MFP zwar gemeldet und er setzt das Pending Bit, sie werden allerding nicht der CPU gemeldet also kein Vektor generiert, was nichts anderes heißt, als daß der Interrupt nicht ausgeführt wird. Ein gelöschtes Bit in diesem Register maskiert den Interrupt aus.

Das Interrupt Mask Register B

$fffa15 B %nnnnnnnn RW imrb

Belegung siehe Interrupt-Enable-Register B
Funktion siehe Interrupt-Enable-Register B

Das Interrupt Vektor Register

$fffa17 B %vvvvsXXX RW vr

Mit dem '**s**' Bit kann man zwischen Sowtware End of Interrupt oder Automatic End of Interrupt umschalten. Es ändert sich dabei das Verhalten des MFP auf den Interrupt. Die '**vvvv**' Bit bilden das obere Nibble für die Vektortabelle des MFP, das untere bildet sich aus dem Level (Prirorität) des Interrupts, der Werte von Null bis fünfzehn annehmen kann.

vvvv = oberes Nibble der Interruptvektoren
s = wenn gesetzt, Software EOI sonst Automatic EOI

Das Timer A Control Register

$fffa19 B %XXXXnnnn RW tacr

nnnn, Bit Nr.
3210

0	0000	TIMER gestopped	
1	0001	Delay Mode,	Vorteiler 4
2	0010	Delay Mode,	Vorteiler 10
3	0011	Delay Mode,	Vorteiler 16
4	0100	Delay Mode,	Vorteiler 50
5	0101	Delay Mode,	Vorteiler 64
6	0110	Delay Mode,	Vorteiler 100
7	0111	Delay Mode,	Vorteiler 200
8	1000	Event Count Mode, Zählen von Ereignissen	
9	1001	Pulsbreitenmessung,	Vorteiler 4
10	1010	Pulsbreitenmessung,	Vorteiler 10
11	1011	Pulsbreitenmessung,	Vorteiler 16
12	1100	Pulsbreitenmessung,	Vorteiler 50
13	1101	Pulsbreitenmessung,	Vorteiler 64
14	1110	Pulsbreitenmessung,	Vorteiler 100
15	1111	Pulsbreitenmessung,	Vorteiler 200

Im Delay Mode kann über einen Vorteiler die am MFP anliegende Taktfrequnez von 2,4576 Mhz geteilt werden und als Interruptquelle genutzt werden.

Im Event Count Mode kann man Ereignisse wie z.B. Frame End beim DMA Sound als Interruptquelle benutzen.

Mit der Pulsbreitenmessung kann man die Zeit zwischen einem Flankenwechsel am TAI Eingang messen.

Das Timer B Control Register

$fffa1B B %XXXXnnnn RW tbcr

Funktion wie bei Timer A Control Register

Das Timer C Control Register

$fffa1D B %XcccXdddX tacr

Näheres siehe Timer A Control Register

ccc und ddd, Bit Nr.

	210		
0	000	TIMER gestopped	
1	001	Delay Mode,	Vorteiler 4
2	010	Delay Mode,	Vorteiler 10
3	011	Delay Mode,	Vorteiler 16
4	100	Delay Mode,	Vorteiler 50
5	101	Delay Mode,	Vorteiler 64
6	110	Delay Mode,	Vorteiler 100
7	111	Delay Mode,	Vorteiler 200

Das Timer A Data Register

$fffa1f B %nnnnnnnn RW tadr

Der im Timer A Data Register enthaltenen Wert wird beim Auftreten eines Ereignisses, z.B. wenn die angegebene Delay Zeit abgelaufen ist, um eins herunter gezählt. Wenn das Data Register von eins auf Null wechseln würde wird im Regelfall der Interrupt ausgelöst und der ursprüngliche Wert im Data Register wieder hergestellt. Wenn der Timer gestartet ist und ein Wert ins Data Register geschrieben wird, wird dieser beim nächsten Durchlauf Startwert. Bei gestopptem Timer wird der ins Data Register geschriebene Wert sofort übernommen.

Das Timer B Data Register

$fffa21 B %nnnnnnnn RW tbdr

Funktion wie Timer A Data Register

Das Timer C Data Register

$fffa21 B %nnnnnnnn RW tcdr

Funktion wie Timer A Data Register

Das Timer D Data Register

$fffa21 B %nnnnnnnn RW tddr

Funktion wie Timer A Data Register

Die Erklärung der MFP-USART Register entfällt, da die serielle Schnittstelle beim Falcon030 mit dem SCC (Serial Communications Controller), der auch in TT/F030 für die LAN (Local Area Network) Schnittstelle zuständig ist, realisiert wurde, die direkte Programmierung dieser Schnittstelle also nicht auf jedem Rechner so funktionieren würde. Außerdem ist diese Schnittstelle für das Programmieren von Spielen auch nur in seltenenen Fällen (z.B. Koppeln zweier Rechner bei Flugsimulatoren) interessant.

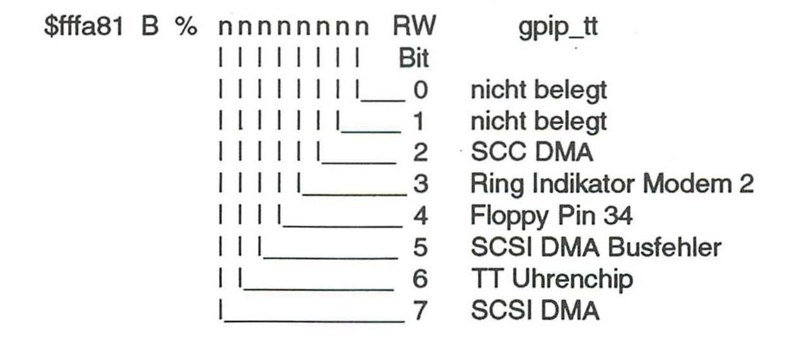
3.2.2 Die Hardwareregister des TT - MFPs

Der TT hat noch einen zweiten MFP, der allerdings nicht so ohne weiteres zu programmieren ist, da für diesen Zweck die entsprechenden Betriebssystem-Funktionen fehlen. Er wird z.B. für die zusätzlichen seriellen Schnittstellen und die SCSI DMA (meldet Ende der Übertragung eines Blocks) benötigt.

Der TT MFP ist aber auch für Spieleprogrammierer interessant, da der ST - MFP bei einem sauber geschriebenen Spiel doch schon mit digitalem Sound, Farbumschaltungen, Sequenzersteuerung u.v.m. ausgelastet sein kann und man hier und da doch gern noch einen zusätzlichen Timer zur Verfügung hätte.

Alle Erklärungen zu den Funktionen der Register siehe ST - MFP!

Gpip Data Register



Das Active Edge Register

$fffa83 B %nnnnnnnn RW aer_tt

Belegung der Bits siehe 'gpip'

Das Data Direction Register

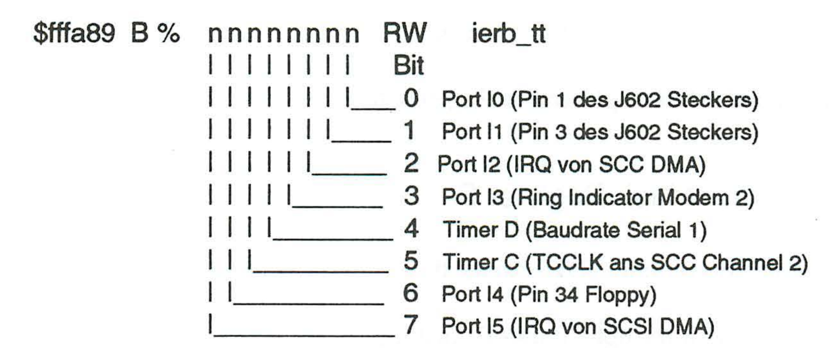
$fffa85 B %nnnnnnnn RW ddr_tt

Belegung der Bits siehe 'gpip'

Das Interrupt Enable Register A

```
$fffa87 B % nnnnnnnn RW     iera_tt
            I I I I I I I I Bit
            I I I I I I I I__ 0 Timer B (Display enable Signal)
            I I I I I I I___  1 XMIT Fehler (RS 232)
            I I I I I I____   2 XMIT Puffer leer (RS 232)
            I I I I I_____    3 RCV Fehler (RS 232)
            I I I I______     4 RCV  Puffer voll (RS232)
            I I I_______      5 Timer A (unbenutzt)
            I I________       6 Port I6 (TT-Uhrenchip/IRQ Anschluß)
            I_________        7 Port I7 (IRQ von SCSI Contr. Chip)
```

Das Interrupt Enable Register B



Das Interrupt Pending Register A

$fffa8B B %nnnnnnnn RW ipra_tt

Belegung siehe Interrupt-Enable-Register A

Das Interrupt Pending Register B

$fffa8d B %nnnnnnnn RW iprb_tt

Das Interrupt In Service Register A

$fffa8f B %nnnnnnnn RW iera_tt

Belegung siehe Interrupt-Enable-Register A

Das Interrupt in Service Register B

$fffa91 B %nnnnnnnn RW ierb_tt

Belegung siehe Interrupt-Enable-Register B

Das Interrupt Mask Register A

$fffa93 B %nnnnnnnn RW imra_tt

Belegung siehe Interrupt Enable Register A

Das Interrupt Mask Register B

$fffa95 B %nnnnnnnn RW imrb_tt

Belegung siehe Interrupt-Enable-Register B

Das Interrupt Vektor Register

$fffa97 B %vvvvsXXX RW vr_tt

Das Timer A Control Register

$fffa99 B %XXXXnnnn RW tacr_tt

Das Timer B Control Register

$fffa9B B %XXXXnnnn RW tbcr_tt

Das Timer C Control Register

$fffa9D B %XcccXdddX tacr_tt

Das Timer A Data Register

$fffa9f B %nnnnnnnn RW tadr_tt

Das Timer B Data Register

$fffaa1 B %nnnnnnnn RW tbdr_tt

Das Timer C Data Register

$fffaa1 B %nnnnnnnn RW tcdr_tt

Das Timer D Data Register

$fffaa1 B %nnnnnnnn RW tddr_tt

3.2.3 Die Vektortabelle der MFP's

Eine Vektortabelle ist eine Liste im Speicher mit lauter Langwörtern, diese Langwörter stehen für Adressen. Diese Adressen werden vom MFP gebraucht, da er dem Prozessor eine Vektornummer sendet, der wiederum erzeugt aus der Vektornummer eine Adresse, in dem er aus der Vektortabelle das Langwort an der Stelle Vektornummer*4 herausholt. Diese Adresse wird angesprungen und hoffentlich eine Interruptroutine ausgeführt. Das passiert allerdings nur, wenn der Interrupt nicht im 'imra' maskiert wurde.

Die ST - MFP Vektortabelle

Adresse	Nummer	Status	Verwendung
niedrigste Priorität			
$100	0	maskiert	Centronics BUSY
$104	1	maskiert	DCD serielle Schnitt.
$108	2	maskiert	CTS serielle Schnitt.
$10c	3	maskiert	GPU Done - Blitter
$110	4	maskiert	Baudr. Gen. ser. Schnitt. (Timer D)
$114	5	aktiv.	200 Hz Timer (Timer C)
$118	6	aktiv.	IKBD/MIDI ACIA's
$11c	7	maskiert	FDC/ASCI DMA - Bus Steuerung
$120	8	maskiert	Display Enable (Timer B)
$124	9	aktiv.	Sendefehler, ser. Schnitt.
$128	10	aktiv.	Sendepuffer leer, ser. Schnitt.
$12c	11	aktiv.	Empfangsfehler, ser. Schnitt.
$130	12	aktiv.	Empfangspuffer voll, ser. Schnitt.
$134	13	maskiert	unbenutzt
$138	14	maskiert	Ring Indicator, ser. Schnitt
$13c	15	maskiert	Monochrome detect
höchste Priorität			

Die TT - MFP Vektortabelle

Adresse	Nummer	Status	Verwendung
niedrigste Priorität			
$140	0	maskiert	I/O Pin 1, J602 Motherboard
$144	1	maskiert	I/O Pin 3, J602 Motherboard
$148	2	maskiert	XDMAIRQ SCC DMA
$14c	3	maskiert	XBRIQ Ring Inicator, Modem 2
$150	4	maskiert	Baudrate Serial 1 (Timer D)
$154	5	maskiert	TCCLK - Takt an SCC (Timer C)
$158	6	maskiert	DCH (Drive Ready) int Floppy
$15c	7	maskiert	XSDMAIRG von SCSI DMA
$160	8	maskiert	Display Enable (Timer B)
$164	9	aktiv.	Sendefehler, ser. Schnitt. TT
$168	10	aktiv.	Sendepuffer leer, ser. Schnitt. TT
$16c	11	aktiv.	Empfangsfehler, ser. Schnitt. TT
$170	12	aktiv.	Empf.puff. voll, ser. Schnitt. TT
$174	13	maskiert	unbenutzt
$178	14	maskiert	TT Uhrenchip
$17c	15	maskiert	SCSI DMA
höchste Priorität			

3.2.4 Betriebssystem-Routinen zum Programmieren des MFP

Hier nur die, die man zum Installieren einer eigenen Interruptroutine benötigt. Achtung, diese Aufrufe funktionieren nur für den ST - MFP.

Nomenklatur:
BYTE = 8 Bit
WORD = 16 Bit
LONG = 32 Bit
VOID = Dummy, d.h. keine Rückgabe
* = Zeiger auf

Rückgabe erfolgt in D0

Beispiel:

XBIOS 31

VOID Xbtimer(WORD timer, WORD control, WORD data, LONG *vec)

ist

```
pea       vec
move.w    #data,-(sp)
move.w    #control,-(sp)
move.w    #timer,-(sp)
move.w    #31,-(sp)
trap      #14
lea       12(sp),sp
```

XBIOS 26

VOID Jdisint(WORD into)

Mit diesem Aufruf ist es möglich, je einen Interrupt des MFP zu sperren.
into, Nummer des zu sperrenden Interrupts 0..15 (siehe 4.2.3)

XBIOS 13

VOID Mfpint(WORD interno, LONG *vektor)

Setzt den Vektor z.B einer eigenen Routine in der Vektor Tabelle.

interno, Nummer des zu überschreibenden Vektors (0 bis 15)
vektor , der zu setzende Vektor

XBIOS 27

VOID Jenabit(WORD into)

Damit kann man einen Interrupt des MFP wieder zulassen.

into, Nummer des zu ermöglichenden Interrupts (0 bis 15)

XBIOS 31

VOID Xbtimer(WORD timer, WORD control, WORD data, LONG *vec)

Diese Funktion installiert einen Timer Vektor, es wird dabei 'vec' in die Vektor-Tabelle geschrieben, das entsprechende Bit im Interrupt Enable und Interrupt Mask Register gesetzt.

timer , Nummer des Timers (0..3 = A..D)
control , setzt das Timer Control Register
data , setzt Timer Data Register
vec , Routine, die bei Interrupt aufgerufen werden soll

3.2.5 Installieren einer MFP Interruptroutine

Dazu gibt es zwei Möglichkeiten: Einmal mittels des Betriebssystems und zum anderen durch direktes Schreiben der Interruptadresse in die Vektortabelle und Zulassen des jeweiligen Interrupts in der Interruptmaske. Das Betriebssystem macht das zwar auch nicht anders, wie wenn man's per Hand in die Register schreibt, aber die Hardware des ATARI's ist keine Konstante, d.h. jedes neue Modell (siehe Falcon030) hat so seine Änderungen, die das Betriebssystem natürlich berücksichtigt (meistens). Andererseits gibt es keine Betriebssystem-Aufrufe, um eine Interruptroutine für den TT MFP zu installieren, also zeige ich beide Möglichkeiten einer Installation.

Installation eines Timer Interrupts via Betriebssystem:

```
pea      interrupt(PC)   ;eigener Vektor auf Stack
move.w   #170,-(SP)      ;170 in Data Register
move.w   #8,-(SP)        ;EVENT COUNT
move.w   #1,-(SP)        ;TIMER B
move.w   #31,-(SP)       ;Xbtimer
trap     #14             ;XBIOS Aufruf
lea      12(SP),SP       ;Stack zurück
```

Installation eines Timer Interrupts "per Hand":

```
pea      Install(PC)     ;Adresse der Installations-
                         ;Routine
move.w   #38,-(SP)       ;Superexec
trap     #14             ;XBIOS Aufruf
addq.l   #6,SP           ;Stack zurücksetzen
```

Install:

```
lea      interrupt(PC),A0        ;Interruptvektor in A0
move.l   A0,$120.w
bset     #0,$FFFFFA07.w          ;Int. Enable A, Timer B zulassen
bset     #0,$FFFFFA13.w          ;Int. Mask A, Maskierung aufheben
move.b   #170,$FFFFFA21.w        ;170 in Data Register
move.b   #8,$FFFFFA1B.w          ;EVENT COUNT, Timer A Ctrl. Reg.
rts
```

Beide Programmteile installieren eine Interruptroutine für den Timer B Interrupt und stellen den Timer in den EVENT COUNT (Zähle Ereignis) mode. Da zählt der Timer B kräftig die Display Enable (DE) Signale, die immer dann kommen (HIGH - 5V), wenn eine neue Zeile dargestellt wird. An den Rändern, wo der Monitor schwarz ist, wird das DE Signal LOW (0V). Nun ist das Bit 3 (Bit 4 für Timer A und Busy an der seriellen Schnittstelle) des Active Edge Register schon von vornherein so eingestellt (gelöscht), daß bei einem Pegelwechsel von High auf Low ein Interrupt ausgelöst wird, so daß der Interrupt am Ende der in Timer B Data Register angegebenen Zeile -1 schon ausgelöst wird, man sich im Zeilenrücklauf befindet. Wenn nun keine anderen Interrupts dazwischenfunken, was sie (wenn man sie nicht abschaltet) aber immer tun, kann man während des Zeilen-Rücklaufs z.B die Hintergrundfarbe ändern und so in jeder Zeile eine andere Hintergrundfarbe zeigen.

Nun noch das Aussehen einer MFP Interruptroutine:

```
interrupt:

 ; hier folgt jetzt die auszuführende Routine
 ; bitte vergessen sie nicht die benutzten
 ; Register (z.B. auf den Stack) zu retten !!!
 ; und vor Beendigung wieder zurückzuschreiben !!!
 bclr #0,$fffffA0F.w     ; nur falls Software End of Interrupt
 rte                     ; Return from Exception !!!
```

Eine jede Interruptroutine (nicht nur die vom MFP ausgelösten) wird mit einem 'RTE' beendet. Das entsprechende Bit (hier Bit 0 / Timer B) im Interrupt in Sevice Bit muß nur dann zurückgesetzt werden, wenn sich der MFP im Software End of Interrupt, die Grundeinstellung nach dem RESET, befindet; dies läßt sich aber mit Bit 3 des Vektor-Registers ändern.

3.3 Abfragen der Joysticks

Da die Joystickdaten der "alten" Ports über den IKBD laufen und dieser einen Interrupt über den MFP auslöst, wenn ein Datenpaket am ACIA anliegt, habe ich mich entschieden, das Thema Joystick in diesem Kapitel zu behandeln.

3.3.1 Abfragen der ST kompatiblen Joystickports

Dies erfolgt durch die Programmierung des IKBD, der muß erst mal so eingestellt werden, daß er Joysticksignale sendet ! Um über das Betriebssystem mit dem IKBD direkt kommunizieren zu können, gibt es zwei Möglichkeiten !

BIOS 3

void Bconout(WORD dev, WORD c)

Gibt Zeichen an ein Ausgabegerät aus.

dev, Nummer des Gerätes
- 0 = parallele Schnittstelle (PRT)
- 1 = serielle Schnittstelle (AUX)
- 2 = VT 52 Konsole (CON)
- 3 = MIDI Schnittstelle (MIDI)
- 4 = Tastatur Prozessor (IKBD)
- 5 = Konsole ohne VT52 (RAWCON)

c , ist das auszugebende Zeichen

XBIOS 25

void Ikbdws(Word cnt, WORD LONG *ptr)

Überträgt eine Zeichenkette zum IKBD
cnt, Menge der zu übertragenen Zeichen -1
ptr , Zeiger auf die zu übertragende Zeichenkette

IKBD auf Joystick Meldungen stellen

```
move.w    #$14,-(SP)   ;IKBD Jystick Ereignisse
move.w    #4,-(SP)     ;Ziel IKBD
move.w    #3,-(SP)     ;bconout
trap      #13          ;bios
addq.l    #6,SP
```

Nun ist das Betriebssystem eigentlich schon für Joystickereignisse ausgelegt, die entsprechenden Pakete werden von den ACIAs abgeholt und irgendwo zwischengespeichert. Allerdings existiert im Betriebssytem keine Routine, die die Joystickdaten verarbeitet. Es gibt aber eine Sprungtabelle in der für jedes IKBD-Ereignis eine Adresse steht, die Routine in dieser Tabell für die Joystickverarbeitung zeigt jedoch auf ein 'RTS' (ReTurn from Subroutine), was soviel bedeutet wir "geh zurück von wo du gekommen bist". Aber das ist kein Problem, wir installieren einfach unsere eigene Routine. Dazu brauchen wir allerdings die Adresse der Vektortabelle und dafür gibt es natürlich auch einen Betriebssystem-Aufruf.

XBIOS 34

LONG Kbdvbase()

gibt Adresse einer Vektortabelle zurück,
die so aussieht:

LONG Adresse von MIDI Eingaberoutine
LONG Adresse von Tastatur Error
LONG Adresse von MIDI Error
LONG Adresse von IKBD Status
LONG Adresse von Mausabfrage
LONG Adresse von Uhrzeitabfrage
LONG Adresse von Joystickabfrage !!!
LONG Adresse von MIDI Systemvektor
LONG Adresse von IKBD Systemvektor
LONG Adresse von IKBD Treiberstatus

Man ersetze nun die Joystickroutine des Betriebssystem durch eine eigene und hole sich die Joystickdaten in einen Speicherbereich, den die eigene Joystickroutine in A0 gemeldet bekommt.
das sieht dann so aus

```
        lea         joy1(PC),A1      ;Adresse wo Daten
                                     ;abgelegt werden
        adda.w      #1,A0            ;Kennbyte überspringen
        move.b      (A0)+,(A1)+      ;Kopieren in Joy1
        move.b      (A0)+,(A1)+      ;Kopieren in Joy2

joy1:   DS.B 1
joy2:   DS.B 1
```

im Byte joy1 und joy1+1 stehen dann die Joystickdaten in Bit 0 bis 3, genau so wie man sie von GFA - BASIC her kennt. In Bit 7 befindet sich der Status des Schußknopfs, bei gesetztem Bit ist der jeweilige Knopf gedrückt.

Jetzt braucht man den IKBD nur wieder zurückzustellen

```
        pea         befehl(PC)       ;IKBD Befehle
        move.w      #1,-(SP)         ;Befehle -1
        move.w      #25,-(SP)        ;Ikbdws
        trap        #14              ;XBIOS
        addq.l      #8,SP

befehl: DC.B $15,8                   ;Joystick aus
                                     ;relative Maus an
```

und den ursprünglichen Joystick-Vektor wieder in die Vektortabelle zurückzuschreiben. Natürlich gibt es ein Beispielprogramm in "KAPB3.TOS" auf der beiliegenden Diskette.

3.3.2 Abfragen 1040 STE kompatiblen Joystickports

Die haben nun ganz und gar nichts mit dem MFP zu tun, aber was solls.

Die "neuen" Joystickports sind wesentlich einfacher abzufragen, da die Joystickdaten auf jeden Fall an der Adresse anstehen und solange anstehen, bis sich die Stellung des Joystick geändert hat. Bei den Paketen des IKBD stehen die Daten nur eine gewisse Zeit in den ACIAs und müssen auch alle abgeholt werden, da man sonst nicht weiß, um welche Art von Daten (Joysticks, Mouse, Keyboard) es sich handelt; man muß also sofort auf die Interrupts reagieren. Man kann bis zu vier Joysticks, zwei Paddles oder einen Light Pen an die neuen Ports anschließen.

Der Schußknopf

$ff9200 W %XXXX XXXX XXXX nnnn R BUTTON

Joystick Ports als Eingang

Spiegelt die Position der angeschlossenen Joysticks wider.

$ff9202 W %oulr oulr oulr oulr W JOY
STICK: 3 1 2 0

o = oben
u = unten
l = links
r = rechts

Joystickport als Ausgang

$ff9202 W %XXXX XXXX aaaa aaaa W JOY

a = beschreibbare Bits

Beim Beschreiben dieser Bits liegt an den Ausgängen bei den gesetzten Bits ein entsprechender 5V Pegel an.

Die Paddels

$ff9210 W %XXXX XXXX pppp pppp RO XPaddle0
$ff9212 W %XXXX XXXX pppp pppp RO YPaddle0

$ff9214 W %XXXX XXXX pppp pppp RO XPaddle1
$ff9216 W %XXXX XXXX pppp pppp RO YPaddle1

pppp pppp, Bits zur Darstellung analoger Werte

Eingang für analoge Joysticks oder Paddles.

Light - Pen/Gun

$ff9220 W %XXXX XXpp pppp pppp RO Xlight
$ff9222 W %XXXX XXpp pppp pppp RO Ylight

Genauigkeit X, 16/Menge der Planes (Falcon ???)
Y, Zeilenweise

Um den genauen X-Wert zu erreichen, muß der Pixel Wert mit einem Faktor multipliziert werden.

Faktor: 16/4*Anzahl der Planes (Falcon ???)

Nun hat es ATARI leider verpaßt, in alle neuen Rechner diese neuen Joystickbuchsen einzubauen, so sind zwar bei den MEGA STE's intern die Buchsen vorgesehen, aber nicht nach außen gelegt worden und auch der TT besitzt keine dieser neuen Anschlüsse. Beim Abfragen dieser Ports muß man also erst mal nachfragen, ob überhaupt welche vorhanden sind. Das kann man zum einem durch Abfragen des Rechnertyps in den Cookies machen und zum anderen durch Abfragen der entsprechenden Register und Abfangen des Bus-Fehlers bei Nichtvorhandensein der Buchsen. Beides birgt einige Probleme in sich, zum einen scheint der TT an der Adresse $ff9200 irgend etwas liegen zu haben, so daß man z.B.

$ff9202 abfragen müßte und zum anderen weiß man nicht, ob in anderen ATARI-Rechnern diese Adressen nicht wieder durch andere Peripherie (wie im TT) belegt wird. Das gleiche Problem tritt auch beim Testen des Rechnertyps per Cookies auf. Ich kann zum einen auf Falcon030 und 1040 STE testen und inoriere kommende Rechnertypen, die möglicherweise auch die neuen Ports besitzen oder ich teste auf 1040 STE und alle Rechener größer als Falcon030 und der Falcon040 (hoffentlich kommt er bald) hat vielleicht gar keine "neuen" Joystickports (wie der TT) mehr.
Nun, wenn ich mich erst einmal entschieden habe, daß der Rechner, auf dem das Spiel läuft, diese Joystickports besitzt, ist die Abfrage kein Problem, einfach im Supervisormode die Speicherstelle in ein Register und den Inhalt testen.

B.4. Sounderzeugung auf den Atari Rechnern

Die Rechner der Atari ST/STE/TT (Falcon ausgenommen) Serie sind lange nicht so flexibel in der Sounderzeugung wie z.B. der AMIGA oder die meisten Soundkarten auf dem PC. Es ist dennoch möglich, mit entsprechendem Knowhow erlesene Soundeffekte diesen Rechnern zu entlocken.

4.1 Grundlagen der Tonerzeugung

4.1.1 Das Wesen der Töne und Geräusche

Töne und Geräusche sind Luftschwingungen. Ähnlich wie man im Wasser Wellenbewegungen z.B. durch Paddeln erzeugen kann, kann man mit einer Stimmgabel die Luft in Schwingungen versetzen. Und ähnlich wie der Seetang am Strand des Meeres von den Wellen hin und her bewegt wird, werden in unsere Ohren kleine Härchen angeregt, mit den Luftwellen zu schwingen. Diese Schwingungen werden dann von Nerven an unser Hirn weitergeleitet und in das umgewandelt, was wir letztendlich hören (das Hirn fungiert dabei wohl als so 'ne Art A/D Wandler ?!). Es gibt also Druck-und Unterdruck-Gebiete, die durch die Schallwellen erzeugt werden.

Wenn zwei Wellen sich treffen, die zum selben Zeitpunkt Druck und Unterdruck haben, verdoppelt sich der Druck bzw. Unterdruck den sie in der Luft erzeugen. Sind zwei Wellen aber gegenläufig, d.h. auf Druck der einen folgt der Unterdruck der anderen, löschen sie sich aus. In einem Lautsprecher werden Wechselströme, die wahrscheinlich auch nichts anderes sind als Schwingungen in den Elektronenwolken des Leiters (z.B. Kupferdraht), somit ähnlich geartet sind wie die Wellen, die wir in Luft (Schall) und Wasser kennen, in Schallwellen umgewandelt. Bei einem normalen ST (ohne E) werden die Wechselströme für den Lautsprecher von einem ärmlichen Chip erzeugt, der nur höchstens drei Rechteckwellen und ein Rauschen hervorbringt. Nun wäre dies schon Stereo (!), wenn es Atari

nicht vorgezogen hätte, alle Tonausgänge zusammenzuschalten. Aber dies ist schon lange her und neuere Geräte haben tatsächlich (viele Jahre nach dem Erscheinen des Amiga) Stereoausgänge.

4.1.2 Kleine Nomenklatur der Sample/Synthesizertechnik

In vielen Programmen werden Ausdrücke wie LFO/Vibrato, Filter, Abspielfrequenzen, Samplingrate ... u.v.m gebraucht. Der "stereotype" Computerfreak, der mit seinem Atari Musik macht und Synthesizer so schnell und so gut wie seinen Radiowecker programmieren kann, weiß natürlich, was die Ausdrücke zu bedeuten haben. Für alle anderen biete ich dieses Kapitel an.

Amplitude - Lautstärke

Je größer die Auslenkung der Wellen wird, desto lauter sind diese.

Filter

Analoge Filter sind Widerstands/Kondensator Glieder, die nur bestimmte Frequenzen durchlassen. Digitale Filter sind dessen rechnerische Nachbildung. Mit digitalen Filtern kann man auch Bausteine erzeugen, die in der Realität nicht möglich sind. Zum Erzeugen solcher digitaler Filter eigenet sich z.B. ein Digitaler Signal Prozessor (z.B DSP 56001), wie er im Falcon eingebaut ist, recht gut. Meist unterscheidet man zwischen zwei Arten von Filtern: Hoch- und Tiefpaß-Filter. Wie schon der Name sagt, läßt die eine Sorte von Filtern hohe Frequenzen ungehindert passieren (Hochpaß) und dämpft die Tiefen beim Tiefpass. Der Tiefpaß verhält sich genau umgekehrt.Nun, bei Geräuschen ist das vielleicht noch verständlich, aber ein Ton hat doch nur eine Frequenz ?

Man kann einen Ton aus vielen Sinuswellen zusammensetzen, den sogenannten Harmonischen (man nennt diese Technik Fouriertransformation). Jede dieser Sinuswellen hat eine andere Frequenz und Lautstärke, wenn man also unendlich viele Sinuswellen hat, die

sowohl in Frequenz als auch in ihrer Amplitude wechseln dürfen, kann man jedes beliebige Geräusch erzeugen. Einige Synthhesizer funktionieren nach diesem System, man nennt dies additive Synthese (sehr schwer zu programmieren).

Filter dämpfen bei allen Wellenformen (außer Sinus) diejenigen Sinuswellen aus denen sie bestehen, die eine bestimmte Frequenz je nach Filtertyp übersteigen oder unterschreiten. Analoge Filter haben grundsätzlich eine bestimmte Flankensteilheit, d.h. sie setzen bei bestimmten Frequenzen langsam ein und haben in einem gewissen Bereich die volle Wirkung. Die Wirkung von digitalen Filtern hängt von deren Rechentiefe ab, je tiefer der Prozessor rechnet, desto näher kommt er dem idealen Filter (es gibt in der Theorie einen idealen Filter, der optimale Ergebnisse liefert. Durch die Toleranzen der Bausteine bei den analogen und der begrenzten Rechentiefe bei digitalen Filtern, ist der ideale Filter nur in der Theorie möglich.).

Resonanz

Nennt man bei synthetischer Tonerzeugung die Rückkoppelung des Filters, dabei wird der Frequenzbereich des Tons angehoben, der an der Grenzfrequenz (CUT OFF) des Filters liegt. Bei vielen Synthesizern lassen sich Grenzfrequenz und Stärke der Rückkoppelung regeln.

Im eigentlichen Sinne ist Resonanz das Mitschwingen und gleichzeitige Aufschaukeln (Verstärken) von Wellen. Zum Beispiel überquert eine Gruppe von Menschen im Gleichschritt eine Brücke. Die Brücke wird in Schwingung versetzt, und wenn das Schwingen der Brücke mit dem Auftreffen der Füße immer wieder zusammentrifft (das ist die Resonanzfrequenz), ist das eine Art von Resonanz. Die Brücke wird in immer stärkere Schwingungen versetzt und kann unter Umständen einstürzen. Wenn eine Saite an einer akustischen Gitarre gezupft wird, schwingt sie und setzt das Gehäuse (Körper) der Gitarre in Schwingungen. Auch das ist Resonanz, denn der Klang der Saite wird ja verstärkt.

Sampling bzw. Digitalisieren im Audio (Ton) Bereich

Als Sampler bezeichnet man im professionellen Musikbereich Geräte, die mittels eines A-D-Wandlers analoge Ton Signale (also so, wie sie in der Natur vorkommen) in digitale (so wie sie der Rechner versteht, im Prinzip 1 und 0) verwandeln und speichern, und diese dann meist mehrstimmig und in unterschiedlichen Frequenzen (Tonhöhen) wieder abspielen können. Diese Geräte sind allerdings immer noch relativ teuer aber liefern meistens eine exzellente Klanqualität d.h. im Bassbereich clean und viel Druck (z.B. für gute Bass Drums) und in den Höhen klar und rein (z.B. für Bells) ohne Nebengeräusche, kein Zischen und kein "Fipsen". Die meisten Rechner kommen bisher an diese Soundqualtität auch nicht annähernd heran, weder Amiga noch irgendwelche gebräuchliche Soundkarten für PC's haben (auch wenn das einige ihrer Besitzer glauben) den Frequenzgang, die Dynamik und den S/N (Signal to Noise - Rauschabstand) professioneller Geräte ("Da is nix mit verdammt nah an der CD !"). Allerdings geht die Entwicklung (Multi Media heißt das Schlagwort) immer mehr in Richtung guter Soundqualität, auch für Computer, während der gehobene Audio Bereich langsam über den Standard der CD hinauswächst.
Also weiter! Aus dem analogen Signal werden Nullen und Einsen, die dann im Speicher des Gerätes gespeichert werden. Je nach A/D (Analog zu Digital) Wandler werden unterschiedliche Pakete von Bits (z.B. Bytes oder Worte) an den Rechner geliefert. Je nach Bitbreite der Pakete, kann man auch von einem anderen Dynamikbereich des Digitizers ausgehen (je höher die Dynamik, desto größer ist der Unterschied zwischen lautestem und leisestem reproduzierbarem Geräusch).

Beispiel: 8 Bit (STE/TT DMA oder AMIGA)
256 Lautstärkestufen

16 Bit (z.B. Falcon030 oder CD)
65536 Lautstärke-Stufen

das läßt sich vergleichen mit der vertikalen Auflösung eines Bildes! Die Horizontale ist dann die Geschwindigkeit in der der Digitizer seine Daten liefert.

Beispiel: ca. 41 kHz (CD Qualität) bedeutet
41000 Abtastungen pro Sekunde
das wiederum bedeutet bei 16 Bit
82000 Bytes Speicher werden in jeder Sekunde gefüllt!

Als Faustregel gilt, daß die halbe Abtastfrequenz die höchste Frequenz ist, die ein Digitizer aufnehmen kann. Das ist auch irgendwie verständlich, da Schwingungen ja aus Änderungen (z.B. des Luftdrucks) bestehen müssen und zu einer Änderung gehören mindestens zwei Werte (ein hoher und ein niedriger Wert). Da der Mensch nur eine geringe Bandbreite (ca. 40 Hz bis 20 kHz) hat, ist es im Normalfall nicht nötig, eine Abtastfrequenz von 41 kHz zu überschreiten. Da man die Dynamik und Abtastrate wegen begrenztem Speicher und Rechenzeit, nicht unendlich hoch wählen kann, bleiben immer relativ hohe Sprünge in den Lautstärkestufen zwischen den einzelnen vom D/A Wandler (wandelt die Bitblöcke in Lautstärkestufen um) abzuspielenden Werten. Um diese zu verringern, kann man entweder die Werte interpolieren und mit einer Art Antialiasing versuchen, die Ecken des Samples abzurunden (digitaler Filter u. ähnl.) oder das analoge Ausgangssignal mit Filterstufen ein bißchen abrunden (oder beides).

Wer sich selbst einen A/D Wandler bauen möchte, um die Sounds in seinen eigenen Spielen zu verwenden, dem empfehle ich in folgenden Ausgaben folgender Computerzeitungen nachzulesen.

Bauanleitungen in:

ST Computer Nr.6 Juni 92, Stereo Sampler für den TT
(auch für ST's und STE's zu gebrauchen)

ST Computer Nr.7/8 August/September 88,
Samples mir's noch einmal Sam

ST EXTRA 3, Das Profiline System

68000er Sonderheft 9, Atari ST als Tonstudio

diese Liste hat keinen Anspruch auf Vollständigkeit und enthält nur die mir bekannten Ausgaben.

Abkürzungen, die in verschiedenen Synthesizern und Musikprogrammen benutzt werden, um die Funktionen anzuwählen:

VC steht immer für Voltage Controlled d.h kontrolliert durch die Spannung.

DC steht für Digital Controlled d.h. gesteuert durch einen Prozessor.

VCF DCF = Regelung des Filters

VCA DCA = Regelung der Amplitude (Lautstärke)

VCO DCO = Regelung der Frequenz

LFO oder Vibrato = Regelung der Tonhöhe über einen zusätzlichen Oszillator meist mit den Wellenformen Dreieck, Rechteck, Sinus, Zufall, erzeugt z.B Effekte wie das Heulen von Sirenen oder die klirrenden Sounds in den Sound - Demos etc.

Envelope, Hüllkurve = mit diesen Parametern läßt sich der Lautstärkenverlauf eines Klangs steuern, z.B. der Sound wird ganz langsam lauter und hört dann abrupt auf oder ist sofort da und klingt langsam aus etc. meist sind Parameter wie Lautstärke und Zeit in verschiedenen Stufen einzustellen.

4.1.3 Soundsynthese

Es ist, wie schon oben erwähnt, möglich, Sounds per Digitizer von außen in den Computer zu befördern. Es ist allerdings auch möglich, Synthesizer zu simulieren und somit die benötigten Sounds, quasi per Hand, einzugeben.

Die Synthesearten, die ich kurz erklären möchte, sind:

ANALOG - gebräuchlich in Geräten wie MINI MOG, Prophet T800, u.v.m.

FM - Frequenzmodulation, gebräuchlich unter anderem in Yamaha Synthesizern.

PD - Phase Distortion (Phase Verzerrung), gebräuchlich in Casio Synthesizern.

RM - Ring-Modulation, wird bei vielen Synthesizern zusätzlich angeboten.

AS - Additive Synthese (Fourier Synthese)

4.1.3.1 Analoge Tonerzeugung

Die allerersten Synthesizer bestanden aus Schwingkreisen als Oszillatoren und RC-(Widerstand, Kondensator) Gliedern als Filter. Da die Bausteine dieser Geräte wärmer wurden, verschoben sich ihre Werte, z.B. wurde der Widerstand kleiner. So ergab sich der schon bekannte "lebendige" Klang dieser Geräte.

Nun kann man diesen lebendigen Klang schlecht simulieren, aber man kann Oszillatoren simulieren, indem man Tabellen mit unterschiedlichen Wellenformen ausliest und an den D/A Wandler übergibt. Ähnlich läßt sich ein Filter simulieren, man muß im Prinzip nur die resultierenden Wellenformen für jede Filterstellung zuvor in Tabellen abgespeichert haben und je nach Filterstellung die entsprechende Tabelle auslesen und an den D/A Wandler übertragen. Wenn man die Abspielfrequenz erhöhen möchte, liest man nicht jeden Wert sondern z.B. jeden zweiten Wert aus der Tabelle und schon hat man die doppelte Frequenz und ist bei einer Oktave höher angelangt.

4.1.3.2 Frequenz-Modulation

Auf diese Technik hat Yamaha das Patent, auch sie ließe sich mit analogen Bausteinen realisieren, mir ist jedoch kein Gerät bekannt, das analoge FM benutzt. Die ersten FM Geräte von Yamaha DX 1/5/7/9/21 und einige Expander (bis TX 81 Z) benutzten 8 Bit D/A Wandler und hatten die für den Hobbymusiker nur schwer in den Griff zu bekommende Rauschfahne. Bei allen folgenden Geräten hat der D/A Wandler mindestens 14 Bit (z.B. DX 7 II oder eben TX 81 Z und DX 11). Diese Geräte läuteten eine neue Ära in der Synthesizer-Technik ein und veränderten das Gesicht der POP/Techno Musik.

Nun die Technik ! "Analog" ließe sie sich in etwa so realisieren, man schließt einen Sinus-Generator mit an den Frequenzeingang eines anderen Sinusgenerators an, so daß die Amplitude des einen zur Frequenz des anderen addiert wird. Da allerdings der Sinus

sich kontinuierlich ändert und man natürlich die Frequenz beider Oszillatoren getrennt ändern kann, erhält man am zweiten Oszillator natürlich keinen Sinus mehr. Das läßt sich natürlich auch mit mehr als nur zwei Oszillatoren bewerkstelligen, und die Lautstärke der Oszillatoren läßt sich kontinuierlich regeln, um einen dynamischen Obertonverlauf (Sinusschwingungen die über der Grundfrequenz liegen) zu bekommen.

Mathematische Formel zur FM für zwei Sinus-Kurven:

sin(5*sin(2x))

Näheres zur FM in dem Buch:

"FM Theory & Applications: By Musicians for Musicians" von Dr. John Chowning und David Bristow, 1986 von der Yamaha Stiftung veröffentlicht.

Wie läßt sich so etwas auf dem Computer realisieren ?

Nun ja, wie immer digital ! Man nehme eine Tabelle von Sinuswerten (übrigens, das geht auch mit anderen Wellenformen !) lese Werte heraus und addiere sie zur Frequenz (Geschwindigkeit), mit der man die Sinuswerte aus einer Sinustabelle liest und immer so weiter (HAHA! rekursiv), je öfter man das macht desto komplexer wird die Wellenform. Die Tabelle mit der Wellenform (z.B. Sinus) besteht je nach verwendetem D/A Wandler aus 8 oder 16 Bit Werten und je nach Format, das der D/A Wandler schluckt, sind diese signed/unsigned.

4.1.3.3 Phase Disortion (Phasenverzerrung)

Diese Technik wird in den Casio Synthesizern der CZ/VZ Serie verwendet. In den Anleitungsheften der Geräte wird die PD Sounderzeugung sehr anschaulich erklärt. Sie beruht im Prinzip auf dem Auslesen von Sinus- oder Cosinus-Wellen aus dem Speicher mit wechselnder Geschwindigkeit. Dadurch erhält man unterschied-

liche Wellenformen. Der Vorteil dieser Art der Soundsynthese besteht darin, daß man keine Filter mehr berechnen muß, da man die Verringerung der Unterschiede in der Lesegeschwindigkeit, dem Sinus bzw. Cosinus immer näher kommt und somit einen Ton mit weniger Obertönen erhält. Natürlich kann man den Unterschied zwischen dieser Art des "Filterns" und einem echten Filter hören, da ein echter Filter nur in Verbindung mit Resonanz, wenn die CUT OFF Frequenz mit der Grundfrequenz übereinstimmt, Sinus-Wellen erzeugen kann.

4.1.3.4 Ringmodulation

Diese Art der Soundsynthese wird in vielen Synthesizern zusätzlich zu ihrer eigentlichen benutzt. Dabei wird ein Welle erzeugt und steuert durch ihren Verlauf die Lautstärke der anderen. Bei zwei Sinus-Wellen würde die Formel dafür etwa so aussehen.

a*sin(s*x)*b*sin(v*x)

a u. b ist die Lautstärke
s u. v ist die Frequenz

Bei vielen Geräten sieht die Formel jedoch so aus:

a*sin(s*x)*b*sin(v*x)+b*sin(v*x)

Nun, eine Formel ist zwar immer eindeutig aber für die Nicht-Mathematiker unter uns nicht unbedingt leicht zu verstehen. Im Prinzip beruht die Ringmodulation darauf, daß die Wellenform des einen Oszillatoren, die Lautstärke des anderen steuert. Bei den alten Synthesizern, bei denen man die Oszillatoren mit eigenem Ein- und Ausgang hatte, war diese Technik der Soundsynthese mit einem zusätzlichen Baustein erledigt, der die Aufgaben eines Wellen-Multiplikators und ggfs. Addierers übernahm. Die digitale Version des Ringmodulators findet man in vielen Casio- und Roland-Synthesizern wieder.

Auf dem Atari kann man ohne großen Aufwand die Ringmodulation selbst ausprobieren, indem man einen Ton auf dem Soundchip (PSG,der mit den "tollen" Rechteck-Wellen) abspielen läßt und dann die Envelope-Geschwindigkeit auf Werte, die im hörbaren Bereich liegen, stellt und auf eine Form stellt, die sich dauernd wiederholt. Durch etwas Herumprobieren erreicht man "schnell" ein vielfaches der Frequenz des Tones, der somit ringmoduliert wird, was wiederum recht interessant klingt. Siehe dazu auch das Listing der Abspielroutine für die Musik zu Argon 4.

4.1.3.5 Additive Synthese

Wie schon erwähnt, kann man sich jede Wellenform als aus vielen Sinuswellen mit unterschiedlicher Frequenz und Amplitude zusammengesetzt vorstellen. Das Zerlegen einer Wellenform in ihre Bestandteile nennt man Fourieranalyse, das schlagen sie besser in einem guten Physik-Buch , im ST Magazin 8-10/88 oder in einem Buch über Signal Processing nach. Aber was man zerlegen kann, kann man meist auch wieder mit dem umgekehrten Verfahren wieder herstellen. Man nehme einige Sinuswellen mit der richtigen Frequenz und Amplitude, überlagere sie (bzw. addiere die Sounddaten, wenn man es numerisch mit dem Rechner löst) und schon erhält man eine schöne "neue" Welle. Natürlich beschränkt sich das Zusammensetzen auf eine endliche Menge von Sinuswellen, was allerdings nicht weiter schlimm ist, da man im Prinzip die Sinuswellen (Harmonischen oder Oberwellen) über einer bestimmten Frequenz nicht mehr wahrnimmt und wir machen ja keine Musik für Hunde, wenn man allerdings einen natürlichen Klang komplett synthetisieren wollte, benötigte man unendlich viele Sinuswellen ("*Ham we nich !*").

Aber grau ist alle Theorie, kommen wir nun zur Realisierung !

4.2 Der PSG (Programmable Sound Generator)

Als die ersten Atari ST's 1985 verkauft wurden, war den meisten, die auf diesem Gerät Spiele schrieben, noch relativ unklar was aus dem Gerät herauszuholen war. Gerade in Sachen Sound hatte man Vieles für unmöglich gehalten, was dann doch von irgendeinem versierten Programmierer erreicht wurde. Zu Beginn, bis ca. 1987 dachten die meisten ST-User noch, es wäre unmöglich, digitalisierten Sound abzuspielen und gleichzeitig Grafik zu bewegen. Als dann einige Spiele und Demos zeigten, daß es doch möglich ist - wer erinnert sich nicht an die LOVE SPY Demo der TNT Crew - war der Bann gebrochen und es gab kaum noch Programme ohne. Das nächste Problem stellte mehrstimmiges Abspielen digitalisierter Sequenzen dar, da der Amiga mit dem Erscheinen des 500er immer mehr Zulauf erhielt und man doch gerne die Spiele mit Sound in annähernd Amiga-Qualität spielen wollte. Spätestens, nachdem man die Cuddly Demo der Carebears gesehen hatte, war klar, daß vierstimmiger Sound mit verschiedenen Abspielfrequenzen, trotz des Yamaha- bzw. General Instument- (rechteck-piepsenden-minimal) Soundchips aus den Urzeiten der Automaten/ Telespielezeit, erlesene Soundereignisse auf dem ST möglich sind, allerdings steht bei angemessener Abspielfrequenz die Maschine quasi still. Ach ja, ich darf nicht vergessen, daß gleichzeitig zur Kultivierung des digitalen Sounds sich noch einige Programmierer darum bemühten fantastische Klänge direkt aus dem Soundchip ohne über den Umweg, den PSG als D/A Wandler umzufunktionieren. Als State of the Art in Sachen Soundchip Music, würde ich im Moment die Punish your Maschine Demo ansehen, speziell die Musik in von ELECTRA ist exzellent !

Um niemand zu kränken, möchte ich noch TEX erwähnen, die mit ihren ersten Sound Demos gezeigt haben, was in Sachen Soundchip-Musik möglich ist.

4.2.1 Die Register des PSG

Der Zugriff auf den PSG erfolgt nur über zwei Adressen, wobei je nach Zugriff, je nach Adresse unterschiedliche Effekte erzielt werden. So daß über den Zugriff auf diese beiden Adressen, eines der 16 internen Register des PSG angewählt und beschrieben werden kann.

Adresse:	Label:	Label:
$ff 8800	giread beim Lesen	giselect (4 Bit) beim Beschreiben

Beim Beschreiben dieses Registers wählt man eines der 16 Register aus, dessen Inhalt man durch Auslesen des Registers erhält.

$ff 8802 giwrite

Durch Beschreiben dieses Registers schreibt man den Wert in das mit 'giselect' angewählte Register der 16 Register des Sounchips.

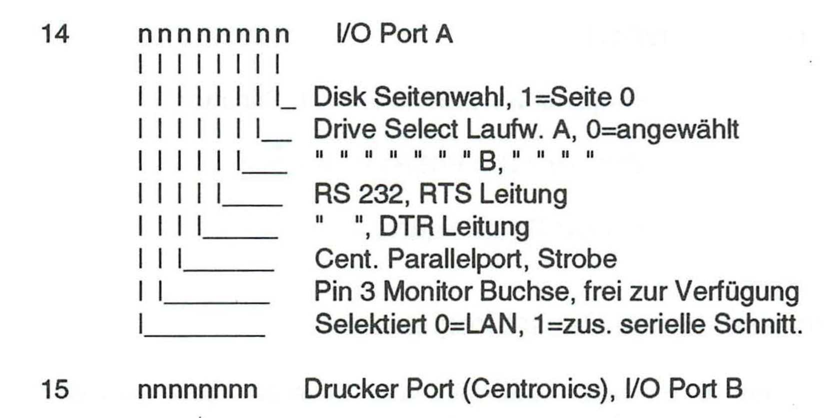
Interne Register des Soundchips:

Wird mit 'giselect' ausgewählt !

```
Nr.  Breite:
0    nnnnnnnn    Per. Dauer  Low  Kanal A
1    XXXXnnnn    Per. Dauer  High       A
2    nnnnnnnn    Per. Dauer  Low  Kanal B
3    XXXXnnnn    Per. Dauer  High       B
4    nnnnnnnn    Per. Dauer  Low  Kanal C
5    XXXXnnnn    Per. Dauer  High       C
6    XXXnnnnn    Per. Dauer  Rauschen
7    nnnnnnnn    MIXER & I/O Port Select
     I I I I I I I I
     I I I I I I I I_ Kanal A       0=AN  1=AUS
     I I I I I I I_   Kanal B       " " " "
     I I I I I I___   Kanal C       " " " "
     I I I I I____    Rauschen A    " " " "
     I I I I_____     Rauschen B    " " " "
     I I I______      Rauschen C    " " " "
     I I_______       I/O Port A  (0=Eingang/ 1=Ausgang)
     I_________       I/O Port B  " " " " " " " " "

8    XXXnnnnn    Lautst.  A unterste 4 Bit
      I
9    XXXnnnnn    " " "    B" " " " "
      I
10   XXXnnnnn    " " "    C" " " " "
      I
      ------ 0=Envelope an  1=Envelope aus

11   nnnnnnnn    Perioden Dauer Envelope Low
12   nnnnnnnn    " " " " " " " " " High
13   XXXXnnnn    Form der Hüllkurve
```



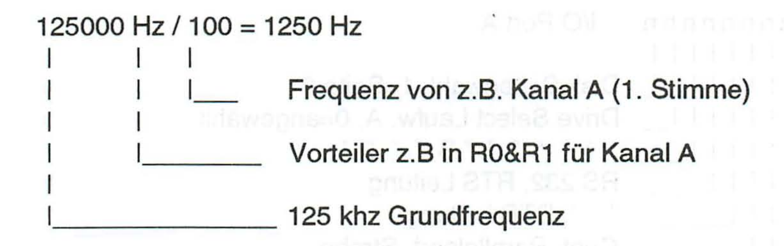
Achtung ! ein undokumentiertes Feature ist, daß die beiden Soundchip Register sich nicht nur an den angegebenen Adressen befinden, sondern sich ab dieser Stelle noch einige Male wiederholen, was es wiederum ermöglicht, gleich mehrere Register des PSG mit einem Befehl zu beschreiben. Siehe dazu auch entsprechende Routine zum Abspielen von "digitalem Sound" im ersten Teil des Buches.

4.2.2 Programmierung des PSG

Nähere Beschreibung der Register:

R0&R1, R2&R3, R4&R5: Tonhöhe

Jeweils zwei Register ergeben einen 12 Bit Wert, der vom PSG als Vorteiler genutzt wird. Da der PSG mit 2MHz getaktet wird und der externe Takt, vom Soundchip durch 16 geteilt wird, ergibt sich eine Frequenz von 125 kHz, die dann auf den 12 Bit Vorteiler treffen. Wenn also der Vorteiler auf dem Wert 100 steht, ergibt sich daraus:



R6 : Vorteiler für die Rauschfrequenz

Diese ist nur 6 Bit breit, was wiederum bedeutet, daß man das Rauschen nicht so fein einstellen kann; dies ist auch nicht unbedingt nötig. Das Rauschen wird durch ein Rechtecksignal mit "zufälliger" Pulsweite erzeugt, d.h. daß die Impulse des Rechtecks, unterschiedlich lang high bzw. low sind, wenn das Muster dieser high/low Signale für das Ohr nicht erkennbar ist, nimmt das Ohr den Ton als Rauschen wahr.

R7 : MIXER und I/O

(Belegung der Bits siehe 4.2.1)

Der Mixer ermöglicht es, den Rauschgenerator zu den einzelnen Tonkanälen dazuzumischen, die Tonkanäle ein- und auszuschalten und die I/O Ports des z.B für das Centronics Interface als Ein- oder Ausgang zu schalten. Wenn man den Centronics Prort als Eingang schaltet, kann man z.B. einen A/D Wandler anschließen und die Daten kontinuierlich von einer Interruptroutine (z.B. im Timer A) abfragen lassen. Als Ausgang programmiert, kann man an den Centronics Port einen D/A Wandler anschließen, was bei entsprechend gutem D/A Wandler guten Sound bedeutet, da man den I/O Port Ausgang in einem Rutsch beschreiben kann und die Lautstärke des Samples nicht in ein für den Soundchip verträgliches Format bringen muß, kann man dabei sogar noch etwas Rechenzeit sparen. Die Bauteilkosten eines einfachen D/A Wandlers, den man an den Centronics Port schließen kann betragen ca. DM 12,-.

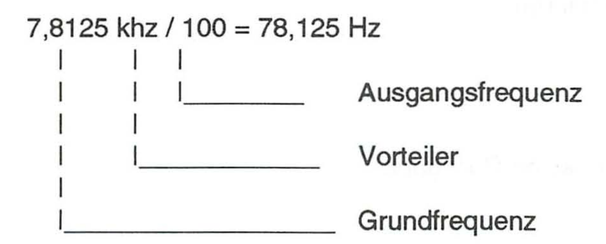
R8, R9, R10 : Lautstärke für die Tonkanäle

Die ersten 4 Bit sind für die Lautstärke je Kanal verantwortlich, das sind 16 Lautstärkestufen, die leider logarithmisch geordnet sind, was das Benutzen des PSG als D/A Wandler etwas erschwert. Trotzdem gibt es mittlerweile einige gute 8 Bit Tabellen, die es ermöglichen, recht guten Sound beim Abspielen der Samples aus dem PSG herauszuholen, trotzdem macht sich in diesem Fall das Fehlen eines Filters durch leichtes Klirren bemerkbar.

Wenn das 5. Bit in einem der Register gesetzt wird, werden die untersten 4 Bit ignoriert, und die Lautstärke von der Hüllkurve gesteuert. Leider gibt es nur einen Hüllkurven-Generator, so daß bei gesetztem 5. Bit bei allen drei Soundkanälen alle drei immer gleichzeitig spielen, sofern sie im Mixer - Register angeschaltet worden sind.

R11&R12 : Periodendauer der Hüllkurve

Das sind zwei 8 Bit Register also 16 Bit, die als Vorteiler für die Geschwindigkeit des Hüllkurvenverlaufs genutzt werden. Eingangsfrequenz ist 2 MHz/ 256, was also 16 mal (!) niedriger ist als die Frequenz, die am Vorteiler für die Tonfrequenz anliegt; das muß man beachten, wenn man die Hüllkurve zur Ringmodulation gebraucht. Bei einer Einstellung von 100:



Das zeigt, daß es bei hohen Tönen Probleme bereitet, den Hüllkurven-Generator für die Ringmodulation zu gebrauchen.

R13 : Form der Hüllkurve

Wenn im Register für die Lautstärke das 5. Bit gesetzt wird, wird die hier angewählte Hüllkurve verwendet. Bei Verwendung der Hüllkurve zur Ringmodulation, wählt man mit R13 die Wellenform, die zur Ringmodulation genutzt wird.

Hüllkurven

0 12
4 13
8 14
9 15
10
11

R14&R15 : I/O Port A,B

Es gehört schon etwas Geschick dazu, mit diesen recht dürftigen Möglichkeiten des PSG wohlklingende Musik aus dem Monitorlautsprecher erklingen zu lassen. Was dazu nötig ist, ist das entsprechende "know how" zum Thema Tonerzeugung.

Über die Betriebssystem-Funktionen 'Giaccess', 'Offgibit', 'Ongibit' und 'Dosound' lassen sich die oben genannten Register über das Betriebssystem programmieren.

BYTE = 8 Bit
WORD = 16 Bit
LONG = 32 Bit
VOID = Dummy, d.h. keine Rückgabe
* = Zeiger auf

XBIOS 28

BYTE Giaccess(WORD data, WORD regno)

Mit diesem Aufruf kann man auf alle Register des Soundchips zugreifen. Also nicht nur Musik machen sondern auch den Druckerport oder das Driveselect Signal manipulieren.

data , Wert der in das Soundchip Register geschrieben wird
regno, Auswählen des Register

Liefert den Inhalt des angewählten Registers zurück

In Assembler:

```
move.w   regno,-(sp)
move.w   data,-(sp)
move.w   #1c,-(sp)
trap     #14
addq.l   #6,-(sp)
```

XBIOS 29

VOID Offgibit(WORD bitno)

bitno, Nummer des zu löschenden Bits im Port A des PSG

XBIOS 30

VOID Ongibit(WORD bitno)

bitno, Nummer des zu setzenden Bits im Port A des PSG

XBIOS 32

VOID Dosound(LONG *psgstr)

Führt ein Programm aus, das den PSG programmiert, damit sind einfache Musikstücke möglich. Die Befehle werden alle 20 ms ausgeführt, die Abspielroutine hängt im Systemtimer und ist somit unabhängig von der Bildwiederholfrequenz (wie andere Soundroutinen).

psgstr, besteht aus einem Feld von Befehlen, die so aussehen können:

Format: Befehlsbyte, Datenbytes

$00 - $0f, Byte 1 (lade in Register)

Bytes 1, in durch Befehlsbyte angegebenes Register laden

$80, Byte 1 (lade in Zwischenspeicher)

Bytes 1, in Zwischenspeicher ablegen

$81, Byte 1, Byte 2, Byte 3 (Schleife)

Zwischenspeicher, Anfangswert für Registerinhalt

Byte 1, Registernummer angeben
Byte 2, wird vom Registerinhalt abgezogen
Byte 3, wenn als Registerinhalt erreicht, Beenden der Schleife
$82-$ff, Byte 1

Byte 1, Inhalt * 20 ms warten, wenn 0, PSG Programm beenden

4.3. Der DMA Sound von STE-TT-Falcon

Den ersten Hinweis auf das DMA Subsystem konnte man schon in den ersten schematischen Zeichnungen über den Aufbau des TT finden. Damals (erste mir bekannte Veröffenlichung 1988) wurde der Sound Shifter genannt. Das erste Gerät, das mit solch einem "Sound Shifter" dann verkauft wurde, war der 1040 STE, allerdings im Videoshifter integriert. Im TT ist der Sound Shifter ein einzelner Baustein, wahrscheinlich, weil mit der Entwicklung des TT Grafikchips schon 1986 begonnen wurde (siehe 68000er Happy Computer Sonderheft Nr. 9) und der DMA Sound zu dieser Zeit noch nicht geplant war.

4.3.1 Das System des DMA Sounds

Begriffserklärung:

DMA = **D**irect **M**emory **A**ccess = Direkter Speicherzugriff

Nun, der Sound - Shifter im TT holt sich immer dann die Daten aus dem Speicher, wenn der Videoshifter eine Pause macht. Das bedeutet, daß der Soundshifter während des Zeilenrücklaufs, das ist immer dann, wenn der Elektronenstrahl am rechten Ende des Bildschirms ist und dann an das linke Ende zurückfährt, seinen 4 Wort großen FIFO (first in first out) Speicher füllt, daher. Das macht er auf demselben Wege wie der Shifter (aus diesem Grund wurde der Sound - Shifter ja auch in den STE's im Shifter untergebracht) da dieser ja während des Zeilenrücklaufs keine Daten benötigt (wird ja auch kein Bild dargestellt). Das bremst den Atari nur minimal, so daß man 50 Khz Stereo Sound abspielen kann, ohne daß man was davon bemerkt.

4.3.2 Die Register zum DMA - Sound

Sound DMA - Control

```
$ff 8900  W %XXXX XXXX XXXX XX  nn RW   sndmactl
                                 | |
                                 | |__ DMA - Sound Enable
                                 |     Bit gesetzt = an
                                 |____ Frame Repeat
                                       Bit gesetzt = Repeat
```

Wird das 1.Bit gesetzt, beginnt die DMA Hardware den Speicherbereich, der in den folgenden Registern angegeben wird, abzuspielen.

Wird das 2.Bit gesetzt, so wird der in folgenden Adressen festgelegte Speicherbereich andauernd wiederholt.

Meine Erfahrung mit dem FALCON030 hat gezeigt, daß man nur das niederwertige Byte (das, in dem die zu setzenden Bits liegen) dieses Registers beschreiben sollte oder besser nur die entsprechenden Bits setzt, da das höherwertige Byte im F030 Einfluß auf den benutzten Interrupt hat, der beim Ende eines abzuspielenden Frames ausgelöst wird.

Der durch folgende Adressen bestimmte Speicherbereich, wird nach der von Atari benutzten Nomenklatur (Namensgebung) Frame genannt. Wahrscheinlich wird das Wort Frame im Sinne von Rahmen oder Eingerahmtem gebraucht, da ja der Speicherbereich quasi durch die beiden Adressen eingerahmt wird.

Frame Startadresse:

High Byte

$ff 8902 W %XXXX XXXX 00nn nnnn RW sndbashi

Mid Byte

$ff 8904 W %XXXX XXXX nnnn nnnn RW sndbasmi

low Byte

$ff 8906 W %XXXX XXXX nnnn nnnn RW sndbaslo

Hier gibt man die Startadresse des Samples ein !

Frame Address Counter:

High Byte

$ff 8908 W %XXXX XXXX 00nn nnnn RO sndadrhi

Mid Byte

$ff 890a W %XXXX XXXX nnnn nnnn RO sndadrmi

Low Byte

$ff 890b W %XXXX XXXX nnnn nnnn RO sndadrlo

in diesen Registern kann man nachlesen, an welcher Stelle im Speicher sich der Soundshifter gerade befindet, ähnlich wie beim Video - Address - Counter im ST.

Frame Endadresse:

High Byte

$ff 890e W %XXXX XXXX 00nn nnnn RW sndendhi

Mid Byte

$ff 8910 W %XXXX XXXX nnnn nnnn RW sndendmi

Low Byte

$ff 8912 W %XXXX XXXX nnnn nnnn RW sndendlo

In diese Register muß die Endadresse des Frames+1! angegeben werden. Der Abstand zwischen den Frames darf nicht kleiner werden als der 4 Wort große FIFO Cache im Sound - Shifter.

Sound Mode Control:

```
$ff 8920 W %XXXX XXXX   n 000 00 n n  RW   sndmode
                        I         I I
                        I
                        I      0 (%00)  6258 Hz
                        I      1 (%01) 12517 Hz
                        I      2 (%10) 25033 Hz
                        I      3 (%11) 50066 Hz
                        I
                        I   Abspielfrequenz
                        I
                        I__%0 = MONO /  %1 = Stereo
```

Wenn der Soundchip auf Stereo geschaltet wird, nimmt er vom gelesenen Wort das High Byte für den linken und das Low Byte für den rechten Kanal. Das Ausgangssignal des DMA - Soundchips wird mittels eines Vier-Stufen-Tiefpass-Filters, dessen Cutoff Frequenz auf 40% der Samplefrequenz geschaltet wird, geglättet.

Wichtig ! Sowohl Anfang/Endadresse sind gepuffert, d.h. sobald der Soundchip begonnen hat, den Sound abzuspielen, kann man die Adressen eines neuen Frames (unter Musikern nennt man das Sequenz) setzen, der Frame wird dann, sofern man 'sndmactl' auf repeat (2.Bit setzen) gesetzt hat, beim nächsten Durchgang gespielt.

4.3.3 Programmierung des DMA - Soundchips

Generell, ist wichtig zu wissen, daß das Format, das der DMA - Soundchip abspielen kann, den Nullpunkt bei "0" hat und nicht bei "127" wie es beim Abspielen mit dem PSG üblich ist.

$80 = -128 = maximale negative Amplitude (Lautstärke)
$00 = 0 = Nullpunkt
$7F = +127 = maximale positive Amplitude

Das Format ist hierbei signed Byte !

Nun, es gibt mehrere Möglichkeiten, den DMA Soundchip zu nutzen.

4.3.3.1 Einmal abspielen des Frames

Das Einfachste ist, Start und Endadresse einer Sequenz in die entsprechenden Register zu schreiben, entsprechende Abspielfrequenz, MONO/STEREO zu wählen und dann das ENABLE Bit zu setzen, um den Sound zu starten. Der Sample (auch 'n Name für Frame) wird dann einmal abgespielt und stoppt. Allerdings ist diese

Technik unbrauchbar zum Abspielen ganzer Musikstücke, da diese bei brauchbarer Abspielfreqenz eine Unmenge von Speicher brauchen.

4.3.3.2 Abspielen unterschiedlicher Frames mit Timer A

Wenn der Timer A sich im Event Count Mode befindet und es sich um einen STE/TT oder Falcon (sofern die Soundhardware entsprechend initialisiert wurde) handelt, dann wird kurz bevor das Sample beendet ist, genau dann, wenn der 4 Wort große FIFO Puffer die letzten Daten eingelesen hat, ein Interrupt ausgelöst. Natürlich nur, wenn das Timer-A-Data-Register auf Null steht, sonst wird der Frame so oft wiederholt, bis das Timer A Data Register auf Null ist. Somit ist es möglich, Sequenzen mehr als nur einmal abspielen zu lassen. Nun, wenn der Interrupt ausgelöst wurde bleibt, je nach Abspielfrequenz, nicht mehr viel Zeit, um die Start/Endadresse in den Registern des DMA - Soundchips zu erneuern, also sollte man dies als allererstes während des Interrupts tun. Man kann sich mit dieser Technik eine Liste der Adressen von abzuspielenden Frames zusammenstellen und zusätzlich eine Liste mit der Anzahl der Wiederholungen, und kann diese Listen per Timer A-Interrupt abarbeiten. Wer es gut machen will, kann auch noch Sprünge zu bestimmten Frames einbauen oder ein Repeat für Blöcke von Sequenzen. Auf jeden Fall verbraucht diese Technik (bei normal großen Frames >20 ms) kaum Rechenzeit, allerdings eine Menge Speicher. Der benötigte Speicher hält sich jedoch gegenüber der ersten Lösung (4.3.3.1) in Grenzen, da ein Musikstück normalerweise so aufgebaut ist, daß z.B. der Refrain mehrmals zu unterschiedlichen Zeiten gespielt wird, aber nur einmal im Speicher gehalten werden muß.

4.3.3.3 Abspielen unterschiedlicher Frames ohne Timer A

Gelegentlich benötigt man den Timer A für alles andere als Musik, wenn man dennoch Sequenzen so abspielen möchte, wie es im Kapitel zuvor beschrieben wurde, muß man eine andere Möglichkeit

finden zu erkennen, wann das Ende eines Frames erreicht wurde. Eine einfache Möglichkeit ist das Abfragen des 'Sound Enable Bits' (1. Bit) im 'DMA - Control Register', denn wenn es gelöscht ist, ist der Sound zu Ende. Das Abfragen dieses Bits erfolgt am besten in einem Interrupt der sowieso schon regelmäßig ausgeführt wird, z.B. Timer C (System Timer) oder VBL, da "hängt" man sich entweder rein oder trägt sich in die entsprechende Liste ein. Um Timingschwierigkeiten zu vermeiden, sollte man sich Sequenzen aussuchen, die an den Schnittstellen (die Stelle, an der das eine Frame an das andere ansetzt) keine Instrumente wie schnelles Becken oder sehr kurze Noten spielen, da an diesen Stellen Veränderungen im Timing besonders auffallen würden. Am besten sind Frames, die in sich abgeschlossen sind. Auch ist es besser, die Samples ein bißchen kürzer abzupielen als sie vom Takt her sein müßten, um die möglichen Verzögerungen vor dem Aufruf des nächsten Frames auszugleichen, denn nichts ist schlimmer für das menschliche Rhythmusempfinden, als wenn ein Beat (Rhythmusschlag) nachhinkt, er sollte lieber etwas früher kommen. Ein bißchen herumprobieren muß man da schon mal !

Eine andere Möglichkeit um herauszufinden, wann der DMA -Soundchip das Ende des Frames erreicht hat, ist durch kontinuierliches Testen der Adress - Counter des Soundshifters. Wobei man dabei, wenn der Soundshifter im Repeat Modus arbeitet eigentlich nach dem Starten der DMA - Soundchips gleich neue Start/Endadressen in dessen Register schreiben muß und dann nur noch testen muß, wann er die neue Adressen angenommen hat, um dann gleich die Adressen des nächsten Frame in die Register zu kopieren. Noch eine Möglichkeit herauszufinden, ob ein Frame zu Ende ist, hat man durch das Abfragen des GPIP des MFP ($fffffa01) da dort das Monochrome-Detect-Bit (Bit 7) und das Frame End Signal mit XOR verknüpft, anliegen. Im Monochrome-Modus (STE/Falcon 640*400 bzw. TT 1280*960) ist das Monochrome-Detect-Bit gelöscht, sonst gesetzt. Wenn der FIFO-Puffer des Soundchips das letzte mal gefüllt wurde, wird dieses Bit Nr. 7 im GPIP in Moncrome gesetzt und das Beschreiben der neuen Start/Endadresse kann

nun erfolgen. Man kann den MFP bei Änderung von I7 einen Interrupt auslösen lassen und so, wie im TIMER A Interrupt, die neue Frame Start/Endadresse eintragen.

4.4. Mehrstimmiges Abspielen von Samples

per DMA und PSG

Wie schon in Programmen zum ersten Teil des Buches gezeigt, ist es auf jedem ATARI (ST-TT-Falcon sowieso) möglich, digitalisierte Sounds mehrstimmig abspielen zu lassen. An dieser Stelle möchte ich erklären, wie es möglich ist, dabei sowohl die Frequenz als auch die Lautstärke jeder einzelnen Stimme zu verändern.

4.4.1 Das Mischen von Samples

Geräusche lassen sich in digitaler Form fast genauso mischen wie in der Natur, durch Addition. Wenn man zwei Geräusche in der Natur mischt, z.B. wenn zwei Personen gleichzeitig sprechen, addieren sich Wellen und Täler. Im Computer übernimmt das Rechenwerk eines Prozessors, in der Regel die CPU - beim Falcon der DSP, der speziell für solche Aufgaben gebaut wurde und das entsprechend schneller kann. Wenn wir nun zwei ziemlich hoch ausgesteuerte Werte miteinander addieren, überschreiten wir den Bereich, den unser D/A Wandler abzuspielen vermag. Eine einfache Lösung ist, wenn man die Werte vor dem Abspielen auf den halben Maximal-Wert bringt, allerdings spielt man ja gelegentlich nur eine Stimme ab, dann wird die einzelne Stimme extrem leise. Aber auch dieses Problem läßt sich lösen, so kann man für eine nicht vorhandene Stimme z.B. einen fixen Wert dazu addieren, ideal ist dabei ein Wert, der beim PSG bei 128 (da dort die Samplewerte im Format unsigned Byte vorliegen), also der X - Achse, liegt.

Im allgemeinen werden aber volle 8 Bit Samples genommen, die einzelnen Werte dieser Samples werden in eine 16 Bit Register-Hälfte addiert und dann um ein Bit nach rechts verschoben (durch

zwei geteilt). Auf diesem Wege hat man zwei Samples miteinander addiert. Natürlich wird, wenn ein Sample länger als der andere ist, nachdem das kürzere Sample beendet ist, statt der vom ausgespielten Sample stammenden Werte der Wert der X - Achse zum noch abzuspielenden Sample addiert.

Einleuchtend ist, daß es nur sinnvoll ist, eine Anzahl von Stimmen zu addieren, die man durch einen Shift-Befehl auch teilen kann, also 2/4/8 ... Stimmen, da eine Division mit dem entsprechenden Assemblerbefehl fast das Zehnfache an Rechenzeit schluckt.

4.4.2 Mehrstimmiges Abspielen mit unterschiedlicher Tonhöhe

Da es selbst auf dem Falcon nicht so ohne weiteres möglich ist, die Abspielfrequenz der DMA - Soundkanäle variabel zu wählen, muß man das, was man abspielt, so modifizieren, daß es sich so anhört als könne man dies.

Aber wie ?

Nichts einfacher als das !?

Beim DMA - Soundchip legt man sich einen Puffer an und strecke bzw. stauche das, was man hineinschreibt.

Hier ein kleines Beispiel für 8 Bit Samples:

in D1 steht der Faktor zum Strecken bzw. Stauchen
in A1 steht die Adresse des abzuspielenden Samples

```
add.l   D1,D2
swap    D2
```

D2 enthält hier im unteren Wort den Offset

```
move.b    0(A1,D2.w),D3
```

In D3 steht das in den Puffer zu übertragende bzw. vom PSG abzuspielende Byte. Es steht in D3, da diese Technik auch beim mehrstimmigen Abspielen verwendet werden kann, also noch ein weiteres Sample addiert werden könnte.

```
        swap    D2
```

Nun, wie funktioniert das genau ?

Wenn z.B. ein sehr kleiner Wert in D1 steht, dann wird das niederwertige Wort gefüllt. Irgendwann gibt es dann einen Übertrag auf das höherwertige Wort. Durch das Tauschen des höherwertigen Wortes mit dem niederwertigen befindet sich im unteren Wort nur der Übertrag. Dieser wird dann zu Adressberechnung genommen und das Byte aus dem Sample geholt. Das funktioniert auch, wenn sich sehr große Werte in D1 befinden (!) die Frequenz läßt sich in einer Auflösung von etwa einem 65536stel einstellen. Das ist eine Art von 32-Bit-Festkommazahlen-Berechnung, wobei sich das Komma immer an der Stelle befindet, wo der Übertrag von einem zum anderen Wort stattfindet.

Mit dieser Technik werden z.B. die unterschiedlichen Frequenzen der 4 Stimmen bei einigen MOD - Playern auf dem ATARI erzeugt.

4.4.3 Lautstärkeregelung beim Abspielen von Samples

Wenn man nun mehrere Samples zusammenmischt, hat man das Problem, das man möglicherweise zwar die Lautstärke des D/A Wandlers regeln kann, z.B. beim DMA - Soundchip von STE/TT/Falcon, aber nicht die Lautstärke jeder einzelnen Stimme.
Eine Möglichkeit die Lautstärke zu regeln ist, sich so viele Tabellen zu erstellen wie man sich Lautstärkestufen wünscht. Wobei die Tabellen so gestaucht werden müssen, daß sie je nach Format (signed oder unsigned)

8 Bit signed:
-128
0 X - Achse
127

8 Bit unsigned:
0
127 X - Achse
255

von der höchsten bzw. niedrigsten zum Nullpunkt hin gestaucht werden müssen !!!

Eine andere Möglichkeit besteht darin, sich eine Tabelle zu schreiben, deren Länge je nach Abstufungen in den Lautstärken und Dynamik gewählt werden muß. Diese Tabelle fällt nicht ganz so schnell ab wie eine 1:1 Tabelle (1 Wert in Tabelle = 1 Wert beim Sample Datum). Wenn man die Lautstärke verschieben will, verschiebt man den X - Achsenschnittpunkt in der Tabelle und addiert eine Konstante zu dem Ausgangswert, um den wieder in dem richtigen Verhältnis zur X Achse zu wissen . Ein Beispiel dazu befindet sich in den Modeplayer Sources.

4.5. Das Falcon DMA - Sound Subsystem

Nun komme ich zu den Änderungen und erheblichen Erweiterungen, die ATARI dem DMA - Sound des Falcon hat angedeihen lassen.

4.5.1 Die Architektur des Falcon030 - Soundsubsystem

Zur besseren Verständlichkeit sehe ich die DMA getrennt von den restlichen Geräten (Devices), weil sie die Verbindung zum Speicher und somit zu CPU den im Speicher abgelegten Klängen etc. darstellt. Sie kann aber wie jedes andere Gerät über die Verbindungsmatrix verschaltet werden.

Das Soundsubsystem besteht aus:

DMA

1) DMA Output Kanal, Speicher nach Device
2) DMA Input Kanal, Device nach Speicher

Diese zwei Kanäle können je bis zu 1 MB/s übertragen und der Prozesssor wird zyklisch unterbrochen, damit die Kanäle ihre 32 Byte FIFOS (Pufferspeicher) füllen können. Diese beiden Kanäle arbeiten (relativ) unabhängig voneinander. Über eine Verbindungsmatrix, mit der es möglich ist, die Devices untereinander und mit den DMA Kanälen zu verbinden, ist es sogar möglich, DMA Input mit DMA Output zu verbinden um somit ähnlich wie beim AMIGA-Blitter Daten von Speicher nach Speicher zu kopieren, ohne daß die CPU auf das Ende des Vorhangs ,wie beim ATARI Blitter, warten muß. Bei hoher Busauslastung z.B. "True" Color Auflösung, kann es passieren, daß die DMA-Kanäle ihre FIFOS nicht mehr schnell genug füllen bzw. in den Speicher übertragen können, was den Verlust von Daten zur Folge hat. Es ist zwar möglich, zwischen DMA/Devices im Handshake (Gerät bestätigt Empfang der Daten und bekommt dann erst weitere gesendet) zu übertragen, es ist mir allerdings nicht gelungen, DMA Input nach DMA Output im Handshake Modus Daten zu schaufeln (wahrscheinlich nicht vorgesehen).

Device Teil

1) External Input/Output, DSP Buchse, Externer serieller Ein-/ Ausgang

2) DSP transmit/receive, Signalprozessor Ein/Ausgang
siehe Kapitel B.5

3) CODEC A/D - D/A Wandler , Kopfhörer/Mikrofon Anschluß

Der CODEC ist ein 16 Bit STEREO Analog/Digital und Digital/Analog Wandler, mit ihm sind sowohl Aufnahme wie auch Abspielfrequenzen bis 50kHz möglich. Über den DMA Input Kanal könnte man allerdings 4 CODEC's parallel mit Daten bei ca. 50kHz betreiben. Alle 4 STERO Sound - Kanäle können nur auf den CODEC abgespielt werden, wenn man diese vorher zusammen mischt, wenn man die DMA Kanäle auf den DSP richtet, könnte dieser das übernehmen und den Sound dann vierstimmig und möglicherweise bearbeitet (Filter, Echo etc.) auf den CODEC geben.

Für den A/D Wandler ist der Mikrofon-Anschluß oder der PSG als Eingang zuschaltbar. Nur auf diese Weise ist der PSG in den F030 integriert, d.h. bei niedriger Abspielfrequenz, die beim CODEC auch immer der Abtastfrequenz entspricht, wird auch der PSG Soundchip Sound entsprechend an Brillianz verlieren (kein großer Verlust !). Den PSG und A/D Wandler gleichzeitig zu Gehör zu bringen ist im F030 nicht möglich.

16 Bit Addierer (wahrscheinlich im CODEC integriert)

Mit ihm ist es möglich, das A/D Signal mit dem D/A Signal zu koppeln, um somit eine Mithören des Eingangssignals (monitoring) zu ermöglichen.

Verbindungs Matrix

Ermöglicht es, eine Quelle z.B. A/D-Wandler mit mehreren Zielen z.B. DMA Kanal in den Speicher zu übertragen.

4.5.2 Programmierung des FALCON - Soundsubsystems

Im Gegensatz zum VIDEO - Subsystem des F030 hat ATARI bei den Aufrufen des Soundsubsystem wirklich ganze Arbeit geleistet, so daß man im Prinzip nicht versucht ist, auf die dazu vorhandene Hardwareregister, die von $ffff8900 bis etwa $ffff893A liegen, zuzugreifen.

Nomenklatur:

WORD = 16 Bit
LONG = 32 Bit
VOID = Dummy, d.h. keine Rückgabe
* = Zeiger auf

Rückgabe erfolgt in D0

Beispiel:

XBIOS 133

VOID settracks(WORD playtracks,WORD rectracks)

ist

```
    move.w   #rectracks ,-(sp)
    move.w   #playtracks,-(sp)
    move.w   #133,-(sp)
    trap      #14
    addq.l    #6,sp
```

XBIOS 128

LONG locksnd()

Sperrt das DMA - Soundsubsystem, so daß keine andere Applikation eine Änderung vornehmen kann.

gibt bei geglücktem Aufruf Null zurück.

sonst -128, wenn das DMA - Soundsubsystem schon gesperrt wurde.

XBIOS 129

LONG unlocksnd()

Gibt das DMA - Soundsubsystem wieder frei.

gibt bei geglücktem Aufruf 0 zurück.

sonst -129, wenn das Soundsubsystem nicht gesperrt war.

XBIOS 130

LONG soundcmd(WORD mode, WORD data)

Setzen der Ausgangslautstärke, z.B. eingebauter Lautsprecher

15 = volle Lautstärke, 240 = geringste Lautstärke

mode: 0 LTATTEN linker Kanal in -1.5db Schritten
data: %XXXX XXXX llll XXXX
Rückgabe: %XXXX XXXX llll XXXX

mode: 1 RTATTEN rechter Kanal in 1.5db Schritten
data: %XXXX XXXX rrrr XXXX
Rückgebe: %XXXX XXXX rrrr XXXX

Setzen der Eingangslautstärke, z.B. Mikrofon Eingang oder PSG

240 = volle Lautstärke, 15 = geringste Lautstärke

mode: 2 LTGAIN rechter Kanal in 1.5db Schritten
data: %XXXX XXXX llll XXXX
Rückgabe: %XXXX XXXX llll XXXX

mode: 3 RTGAIN rechter Kanal in 1.5db Schitten
data: %XXXX XXXX llll XXXX
Rückgabe: %XXXX XXXX llll XXXX

Setzen der Quellen für den 16 Bit Addierer

Wichtig ! Wenn man DMA u. PSG gleichzeitig spielen lassen möchte, müssen wie im STE/TT beide Bits gesetzt sein.

mode: 4 ADDERIN

data:	BIT	7	6	5	4	3	2	1	0
		0	0	0	0	0	0	M	A

A = ADC , (Analog Digital Converter) Mikrofoneingang
M = Matrix , 4*4 Quell - Ziel Matrix

gibt ein Wort mit altem Status zurück

Setzen der Quelle für den Analog Digital Wandler (ADC)

mode: 5 ADCINPUT

data:	BIT	7	6	5	4	3	2	1	0
		0	0	0	0	0	0	L	R

L: Linker Kanal
R: Rechter Kanal

wenn gesetzt PSG, sonst Mikrofon Eingang

Es ist damit möglich, den linken oder rechten Kanal vom Mikrofon oder PSG auf den A/D Wandler zu schicken; dieser Eingang läßt sich dann über den 16 Bit-Addierer mit dem DMA Sound mischen und zum Lautsprecher/Kopfhörerausgang schicken.

gibt ein Wort mit altem Status zurück

Setzen dee STE/TT Vorteilung
Die hier eingestellten Werte werden vom System nur genutzt, wenn bei 'devconnect' der Wert von 'prescale' auf Null gesetzt ist.

mode: 6 SETPRESCALE

DATA:
0 - Nicht möglich d.h. kein Ton
1 - Teiler 640 entspricht 12.5 kHz STE/TT
2 - Teiler 320 entspricht 25.0 kHz STE/TT
3 - Teiler 160 entspricht 50.0 kHz STE/TT

XBIOS 131

LONG setbuffer(WORD reg, LONG begaddr, LONG endaddr)

Zum Setzen der Anfang/Endadresse eines Aufnahme/Abspielpuffers, auf den dann von der Sound DMA zugegriffen werden kann.

reg = 0 abspielen, 1 aufnehmen
begaddr = Startadresse des Puffers
endaddr = Endadresse des Puffers

gibt 0 zurück, wenn kein Fehler aufgetreten ist.

XBIOS 132

LONG setmode(WORD mode)

Voreinstellung der Sound Kanäle, gilt für Aufnahme und Wiedergabekanäle gleichermaßen.

mode, 0 = 8 Bit STEREO (STE/TT compatibel)
1 = 16 Bit STEREO
2 = 8 Bit MONO (STE/TT compatibel)

gibt 0 zurück, wenn kein Fehler aufgetreten ist.

XBIOS 133

LONG settracks(WORD playtracks, WORD rectracks)

Setzt die Menge der Kanäle für Aufnahme und Wiedergabe.

playtracks , 0 bis 3
rectracks , 0 bis 3

gibt 0 zurück, wenn kein Fehler aufgetreten ist.

XBIOS 134

LONG setmontracks(WORD montrack)

Mit diesem Aufruf kann man einstellen, welcher der "vier" Soundkanäle über den CODEC ausgegeben werden, da nur ein D/A Wandler vorhanden ist, kann das immer nur ein Kanal sein.

montrack, 0 bis 3

gibt 0 zurück, wenn kein Fehler aufgetreten ist.

XBIOS 135

LONG setinterupt(WORD src_inter, WORD cause)

Das ist wichtig für die STE/TT-DMA-Sound-Kompatibilität, denn im Gegensatz zu deren DMA - Sound, bei dem am Ende des Puffers sowohl ein Timer A als auch ein MFP I7 Interrupt ausgelöst werden kann, läßt sich beim F030 extra einstellen, welcher Interrupt ausgelöst werden soll, wenn das Ende des Wiedergabe-Puffers und/oder das Ende des Aufnahme-Puffers erreicht ist.

src_inter , 0 = TIMER A, 1 = MFP I7
cause , 0 = kein Interrupt, 1 = Wiedergabe, 2 = Aufnahme
3 = Aufnahme und Wiedergabe

Um STE/TT-Kompatibilität zu erreichen, muß TIMER A und MFP I7 auf Interrupt am Ende des Wiedergabepuffers eingestellt werden.

gibt 0 zurück, wenn kein Fehler aufgetreten ist.

XBIOS 136

LONG buffoper(WORD mode)

Funktioniert im Prinzip wie beim STE/TT DMA - Sound

mode,	BIT	7	6	5	4	3	2	1	0
		0	0	0	0	RR	RE	PR	PE

Bit gesetzt gleich an

PE Wiedergabe
PR dauerndes Wiederholen des Wiedergabepuffers
RE Aufnahme
RR dauerndes Wiederholen des Aufnahmepuffers

Wenn mode = -1, dann gibt die Funktion den Status der Bits zurück. Wenn die 32 Byte FIFO das Ende des Puffers erreichen, schalten die entsprechenden Bits PE und/oder RE kurzzeitig auf 0, selbst wenn das RR und/oder PR Bit (dauerndes Abspielen oder Aufnehmen von einem Puffer) gesetzt ist.
oder gibt 0 zurück, wenn kein Fehler aufgetreten ist.

XBIOS 137

LONG dsptristate(WORD dspxmit, WORD dsprec)

Man kann mit diesem Aufruf den DSP von der Verbindungs-Matrix trennen. Wobei man das Senden als auch das Empfangen des DSP in/von der Matrix separat an/abschalten kann.

dspxmit, DSP sendet in Matrix: 0 = abkoppeln, 1 = möglich
dsprex, DSP empfängt von der Matrix: 0 = abkoppeln, 1 = möglich

gibt 0 zurück, wenn kein Fehler aufgetreten ist.

XBIOS 138

LONG gpio(WORD mode, WORD data)

Dieser Aufruf dient zur Ansteuerung der GP Pins der DSP Buchse.

mode, 0 = setze Ein/Ausgabe Richtung
1 = Lesen
2 = Schreiben

data: %rrrrrrrr rrrrrnnn r = reserviert, n = benutzt

wenn mode =0, dann setzt data die Ein/Ausgabe Richtung der Pins
wenn mode =2, der Inhalt von data bestimmt den Pegel an den GP Pins der DSP Buchse

gibt zurück:

für mode = 1 (lesen), die anliegenden Pegel an den GP Pins DSP Buchse als Bitmuster

sonst 0

XBIOS 139

LONG devconnect(WORD src, WORD dst, WORD srcclk,
WORD prescale, WORD protocol)

Dieser Aufruf, macht 'ne ganze Menge !

Er verbindet eine Quelle mit dem gewünschten Ziel in der Verbindungsmatrix. Wobei die Quelle immer nur ein Device und Ziel mehrere Devices sein können. Außerdem kann man mit dieser XBIOS Funktion den Takt und den Vorteiler für das Sound - Subsystem, sowie das Protokoll zwischen den einzelnen Devices wählen.

src, (source) setze Quelle für die Verbindungsmatrix

0 = DMA Wiedergabe
1 = DSPXMIT (DSP Senden)
2 = EXTINP (Externer Eingang)
3 = ADC (Mikrofon Buchse oder PSG)

dst, (destination) setzt Ziele in der Verbindungsmatrix

Bit 0, DMA Aufnahme
1, DSPRECV (DSP Empfangen)
2, EXTOUT (Externer Ausgang)
3, DAC (Kopfhörerbuchse oder Lautsprecher)

srcclk, was als Takt für CODEC und DMA benutzt wird

0 = 25.175 MHz intern (STE/TT kompatibel)
1 = extern (von der DSP Buchse)
2 = 32 MHz intern

prescale, Vorteiler für den anliegenden Takt

Darf Werte von 0 bis 12 annehmen, Werte größer als 12 schalten das System still, Werte von 1 bis 12 werden als Vorteiler für den Takt/256 verwendet. Wenn der Vorteiler null ist wird der STE/TT kompatibele Vorteiler benutzt, siehe soundcmd (XBIOS 130)

protocol,

Schaltet das Handshaking zwischen den Devices an und aus

0 = Handshaking an
1 = Handshaking aus

gibt 0 zurück, wenn kein Fehler aufgetreten ist.

XBIOS 140

LONG sndstatus(WORD reset)

Mit diesem Aufruf erhält man die momentane Verfassung des CODEC.

Die Rückgabe erfolgt im niederwertigen Nibble. Auch wird zurückgeliefert, ob der rechte oder linke Kanal während der A/D Wandlung und des Filterns übersteuert wurden.

reset:

wenn 1 dann wird das komplette Sound - Subsytem initialisiert.

DSP von Verbindungsmatrix getrennt (tristatet)
Lautstärke Aufnahme u. Wiedergabe auf Null gestellt
Matrix Verbindungen werden zurückgesetzt
ADDERIN - der Addierer wird ausgeschaltet
mode ist 8 Bit Stereo

Aufnahme/Wiedergabe Kanäle werden auf Null gesetzt
Abhörkanal ist Null
keine Interrupts am Pufferende (TIMER A, MFP I7)
keine Puffer gesetzt

gibt den Status des CODEC zurück

Bits 0 bis 3 %SSS

0 = kein Fehler aufgetreten
1 = ungültiges Kontrollfeld
2 = ungültiges Syncformat (CODEC schaltet still)
3 = Takt wird vom CODEC nicht unterstützt (CODEC schaltet still)

Bit 4 & 5 %LR

R = rechter Kanal übersteuert
R = linker Kanal übersteuert

XBIOS 141

LONG buffptr(LONG *pointer)

Diese Funktion gibt die "momentane" Position der Aufnahme, Wiedergabe DMA in den Puffern zurück.

*pointer, Zeiger auf vier Langworte in denen nach dem Aufruf folgendes steht

1. LONG, DMA Position im Wiedergabe-Puffer
2. LONG, DMA Position im Aufnahme-Puffer
3. LONG, reserviert
4. LONG, reserviert

gibt 0 zurück, wenn kein Fehler aufgetreten ist.

Hier die 'prescale' Tabelle für 'devconnect' !

Wert	Frequenz
0	STE/TT kompatibler Vorteiler
1	49170Hz
2	33880Hz
3	24585Hz
4	20770Hz
5	16490 Hz
6	14285 Hz CODEC nicht geeignet für diese Frequenz
7	12292 Hz
8	11110 Hz CODEC nicht geeignet für diese Frequenz
9	9834 Hz
10	9090 Hz CODEC nicht geeignet für diese Frequenz
11	8195 Hz
12	7690 Hz
13	7140 Hz
14	6660 Hz
15	6250 Hz

Leider kann es zu Problemen kommen, wenn XBIOS Aufrufe aus einem Interrupt erfolgen, da das XBIOS nur beschränkt reintrant ist, das bereitet größere Probleme beim DMA - Sound Subsystem, da ja dessen Hardwareregister nicht dokumentiert sind, man aber nur über XBIOS Aufrufe die DMA Puffer setzen kann. Wenn also am Ende des Puffers ein TIMER A oder ein I7 Interrupt erfolgt, kann man nicht unbedingt in dieser Interruptroutine einen XBIOS-Call absetzen, man muß zuvor einige Vorkehrungen treffen.

Zu diesem Zweck gibt es eine Systemvariable, 'savptr' an der Adresse $4a2, die den Zeiger auf die Adresse des Speicherbereichs enthält, in dem XBIOS und BIOS Registerinhalte, Rücksprungadressen etc. gerettet werden. Beim Eintritt in den Interrupt sollte man, wenn man eine XBIOS/BIOS Funktion aufrufen will, den in 'savptr' enthaltene Zeiger um 46 Bytes heruntersetzen und vor der Beendigung der Routine wieder zurücksetzen. So darf sich immer nur eine Interruptroutine verhalten und es ist nur Platz für ca.

drei Rekursionen der Interruptroutine d.h. der Zeiger darf nicht mehr als 138 Bytes nach unten geschoben werden, für mehr ist eben kein Platz.

4.6 Das Microwire Interface (nur STE/TT)

Da hat Atari einen kleinen Equalizer springen lassen !

Im Prinzip ist das Microwire Interface nur der Chip, der den kleinen Equalizer (LMC1992) anspricht und zwar in einer Geschwindigkeit von 16×10^{-6}s. In ST-TT ist dieses Interface nur dazu da, um den LMC1992, anzusprechen. Es wäre allerdings möglich, auch andere Geräte über dieses Interface anzusprechen, warten wir also auf die zukünftigen Erweiterungen ATARI's. In den F030 Entwicklergeräten war das Microwire Interface noch vorhanden, bei den Seriengeräten fiel das Interface dem Rotstift zum Opfer. Allerdings läßt sich die Klangregelung des LMC1992 auch gut mit Falcon's DSP 56001 (Signal Prozessor) nachahmen, aber dazu mehr in Kapitel 7, das sich mit dem DSP beschäftigt.

4.6.1 Die Hardware - Register des Microwire Interface

Adress+Data Bits

$ff 8922 W %nnnn nnnn nnnn nnnn RW MWDATA

Adress+Data Bits Bedeutung:

Bits:

10-11 Adresse des Gerätes (Device 10 = LMC1992)

restlichen Bits sind Daten, die der LMC so interpretiert:

Bits:

7-9 MODUS restlichen Bits:

000 Mischung einstellen

```
XXX X nn
      | |
      00 -12 dB
      01 mische PSG zum Ausgang
      10 mische PSG nicht zum Ausgang
      11 reserviert
```

001 Baß (50 Hz)
010 Höhen (15 kHz)

```
XX  n nnn
    | |||
    0 000 -12 dB
    ...
    ...
    0 110 Mitte
    ...
    ...
    1 100 +12 dB
```

```
100   rechte Lautstärke
101   linke Lautstärke

          X   nn  nnn
              ||  |||
              00  000 -40 dB
              ..  ...
              ..  ...
              01  010 -20 dB
              ..  ...
              ..  ...
              10  1XX  0 dB


 011  Gesamt-Lautstärke

          nnn  nnn
          |||  |||
          000  000 -80 dB
          ...  ...
          ...  ...
          010  100 -40 dB
          ...  ...
          ...  ...
          101  XXX  0 dB
```

Mask-Register

$ff 8924 W %nnnn nnnn nnnn nnnn RW MWMASK

Wenn das Data Register beschrieben wird, wird das Mask Register geshiftet; es dauert eine gewisse Zeit, bis das Register neu beschrieben werden kann, denn der Inhalt des Data - Registers wird hinausgeschoben und damit quasi in das angegebene Gerät (z.B. %10 LMC1992) übertragen. Gleichzeitig wird das Mask-Register gerollt; wenn das Mask Register seinen alten Zustand hat, sind alle

Bits des Data Registers übertragen worden, allerdings muß dafür das Mask - Register eine Bitkombination enthalten; die nur wenn sie ganz "durchgerollt" wurde; wieder die ursprüngliche Zahl ergibt z.B. $07FF.

4.6.2 Einfaches Nutzen des LMC1992

Voraussetzung:

Letzte Übertragung geschah genauso (also alle Bits schon durchgeschoben).

```
MWMASK    equ $ffff8924
MWDATA    equ $ffff8922

  move.w   #$07FF,MWMASK.w              ; Nur beim ersten Zugriff
  move.w   #%10000000010,MWDATA.w       ; PSG wird nicht zum
                                        ; Ausgang dazu gemischt
  cmp.w    #$07ff,MWMASK.W
```

Erklärung:

Beim allerersten Zugriff schreibt man #$07ff in das Mask -Register, dann kann man das gewünschte Datum in das Datenregister schreiben z.B. %10 000 000 010

```
            |    |        |
      DEVICE SET-MIX  kein PSG zum Ausgang
```

bei jedem weiteren Zugriff ist es im Prinzip nicht nötig, das Maskenregister neu zu beschreiben, denn es behält ja seinen Inhalt. Jedoch kann es durchaus passieren, daß jemand einen anderen Wert in das Maskenregister hinein schreibt, z.B. ein parallel laufendes Programm (unter MINT oder MULTI TOS).

4.7. Das Amiga Sound - Modul (MOD) Format

Offset

$00 - $13	Name des Moduls
$14 - $3b5	31 je 30 Byte große Einträge für maximal 31 Samples

Aufbau der Einträge:

$00 - $15	Name des Samples
$16	Länge des Samples/2
$18	Lautstärke des Samples ($00-$40)
$20	Loop Beginn von Sampleanfang Offset/2
$22	Looplänge/2

$3b6	Anzahl der Patterns
$3b7	Frei
$3b8	Patternliste, gibt an, in welcher Reihenfolge die Patterns abgespielt werden.
$438	Zeichen des Soundtrackers, mit dem das 'MOD' erstellt wurde
$439	Noten 1024 Byte (4 Noten * 64 pro Pattern) in der Reihenfolge:

1. Note Stimme 1
1. Note Stimme 2
1. Note Stimme 3
1. Note Stimme 4

usw.

anschließend folgen die Samples

Format einer Note:

N = Nibble

```
N NNN    N N NN
|   |    |  |  |_____ Intensität des Effekts
|   |    |  |
|   |    |  |_________ Effekt
|   |    |
|   |    |___________ Nr. des Samples
|   |                 Low Nibble
|   |
|   |_______________ Tonhöhe (Periode)
|                     (Amiga Hardware spezifisch)
|
|____________________ Nr. des Samples
                      High Nibble
```

Format der Tonhöhe:

Perioden - Werte

	1. Oktave	2. Oktave	3. Oktave
C	$ 3 58	$ 1 AC	$ 0 D6
C#	$ 3 28	$ 1 94	$ 0 CA
D	$ 2 FA	$ 1 7D	$ 0 BE
D#	$ 2 D0	$ 1 68	$ 0 B4
E	$ 2 A6	$ 1 53	$ 0 AA
F	$ 2 80	$ 1 40	$ 0 A0
F#	$ 2 5C	$ 1 2E	$ 0 97
G	$ 2 3A	$ 1 1D	$ 0 8F
G#	$ 2 1A	$ 1 0D	$ 0 87
A	$ 1 FC	$ 0 FE	$ 0 7F
A#	$ 1 E0	$ 0 F0	$ 0 78
H	$ 1 C5	$ 0 E2	$ 0 71

Effekte:

(entnommen aus der Anleitung zu NoiseTracker 1.1)

0 - arpeggio
1 - portamento up
2 - portamento down
3 - Tone-portamento
4 - Vibrato

A - Slide volume
B - Position jump
C - Set volume
D - Pattern break
E - Set filter
F - Set speed

$0 Arpeggio - $0 + zweiter & dritter Halbton über der Note
Erzeugt einen Akkord mit nur einer Stimme

C-3 00037 erzeugt darüber liegenden Akkord
C-3 00047 erzeugt darunter liegenden Akkord

$1 Portamento up - $1 + Geschwindigkeit
Erhöht langsam die Tonhöhe
C-3 00103 1 ist der Befehl, 3 die Geschwindigkeit.

$2 Portamento down - $2 + Geschwindigkeit
Läßt die Tonhöhe fallen
C-3 00203 2 ist der Befehl, 3 ist die Geschwindigkeit.

$3 Tone-portamento - Ziel Note + $3 + Geschwindigkeit
Langsames Verändern von der Tonhöhe der letzten Note zur kommenden.
C-3 00305 C-3 Note, zu der erhöht wird, 3 der Befehl, 5 die Geschwindigkeit.

$4 Vibrato - $4 + Vibratogeschwindigkeit + Vibratospanne

C-3 00481 4 der Befehl, 8 die Vibratorgeschwindigkeit, 1 Vibratodynamik

Um das Vibrato fortzusetzen, muß man nur die 4 in der nächsten Note setzen.

$A Volume-slide - $A + Geschwindigkeit nach oben bzw. unten

C-3 00A05 5 Geschwindigkeit nach oben
C-3 00A40 4 Geschwindigkeit nach unten

$B Position-jump - $B + Position an der Fortgesetzt wird

C-3 00B01 1 ist die Position an der das Lied fortgesetzt wird.

Das laufende Pattern wird unterbrochen !

$C Set volume - $C + neue Lautstärke
Setzt die Lautstärke bis $40 möglich

C-3 00C10 C ist der Befehl, 10 die Lautstärke.

$D Pattern-break - $D + nothing
Fährt fort mit dem nächsten Pattern

C-3 00D00 D als Befehl, Rest egal

$E Set filter - $E + Filter an/aus
Man kann mit diesem Befehl auf einigen Amiga 500/2000er den Filter ein/ausschalten (auf dem STE/TT könnte man das mit dem Microwireinterface simulieren, auf dem Falcon sicherlich mit dem DSP 56001)

C-3 00E01 Filter aus und LED aus
C-3 00E00 Filter an und LED an

$F Set speed - $F + Geschwindigkeit
Ändert die Geschwindigkeit, in der der Song abgespielt wird, Geschwindigkeiten $01 - $1f sind erlaubt
C-3 00F07 stellt die Geschwindigkeit auf 7 ein
d.h. alle 7*20ms wird eine neue Note gespielt.

Die Wirkungen der Effekte werden alle 20ms geändert !

B.5. Das Ansteuern von Klangerzeugern per MIDI Schnittstelle

Wer schon mal eines der Sierra Adventures mit über MIDI angeschlossenen MT-32 gespielt hat, weiß, daß die Musik, die man da hören kann, schon fast dem Soundtrack eines Films gleich kommt. Insofern ist diese MIDI - Schnittstelle für Spieleprogrammierer schon eine sehr interessante Sache. Leider machen sich nur wenige Firmen die Mühe, dieses Feature in ihren Programmen zu nutzen. Obwohl das Implemetieren der MIDI Abspielroutinen in vielen Fällen nur ein geringfügiger Mehraufwand bedeuten würde. Auch haben die wenigsten exzessiven Spieler, also im Prinzip der Markt der Spielehersteller einen MIDI-fähigen Klangerzeuger. Also warum, denkt sich der Programmierer, soll man sich die Mühe machen etwas anzusteuern, was eh keiner besitzt. Der Konsument sieht in diesem Fall natürlich nicht ein, wieso er sich einen Expander (Synthesizer ohne Tasten) kaufen soll, wenn es kein Spiel gibt, das so ein Teil unterstützt (eine verfahrene Sache).

Was ich damit sagen will, man sollte sich gelegentlich überlegen, in Demos und Spielen mit viel Musik MIDI zu unterstützen.

5.1 Was ist MIDI ?

MIDI (Musical Instrument Digital Interface also eine digitale Musik-Instrumenten-Schnittstelle) ist eine genormte Schnittstelle zwischen Musik-Instrumenten, Sequenzern, Computer, Mischpulten und vielem mehr. Irgendwann zu Beginn der 80er Jahre haben sich namhafte Hersteller von Synthesizern und ähnlichem hervoragendem Gerät zusammengesetzt um eine Schnittstelle zwischen ihren Erzeugnissen zu entwerfen. Diese Schnittstelle hat eine maximale Übertragungsrate von 31 KBit, das sind fast 4 Kbaud (Kilobyte pro Sekunde). Aber auch das ist bei großen Geräteparks manchmal zu wenig, und es kommt zu den sogenannten "MIDI delays" , d.h. das erste Gerät in der MIDI-Schlange spielt die Note merklich früher

als das letzte. Die MIDI-Daten werden alle byteweise übertragen (8 Bit, getrennt durch Start- und Stopp-Bit), auch sollten die MIDI - Verbindungskabel nicht länger als 15 Meter sein.

So, das war das Formelle, jetzt kommen wir zur ...

5.2. Allgemeine MIDI Nomenklatur

5.2.1 Was ist ein MIDI Kanal ?

Zu jedem Befehl, der über MIDI gesendet wird, wird noch der MIDI-Kanal dazugesendet um sicherzustellen, daß er auch vom richtigen Gerät oder richtigem Teil eines Gerätes empfangen wird. So ist es möglich, bis zu 16 verschiedene Geräte über einen MIDI Ausgang zu steuern. Natürlich muß man zuvor auch die entsprechenden Kanäle an den Geräten eingestellt haben. Den MIDI Kanal kann man sich wie die Telefonnummer eines Gerätes vorstellen, nur das Gerät, das die richtige Nummer hat nimmt auch die entsprechenden Daten an. Wenn zwei Geräte dieselbe Nummer haben, spielen sie auch beide dieselben Noten.

5.2.2 Unterschiedliche MIDI Modes

OMNI   Empfang auf allen MIDI Kanälen gleichzeitig
POLY   mehrere Stimmen gleichzeitig
MONO   nur eine Stimme

OMNI ON, POLY

Das Gerät empfängt auf allen MIDI Kanälen und spielt die empfangenen Noten alle ab. Ältere Geräte, wie die ersten Poly 800 (ließen sich mit neuem Eprom umrüsten) besitzen diesen MIDI Mode, in einigen Geräten läßt er sich einstellen. In einem System mit mehreren Geräten kann man ein Gerät mit dieser Einstellung nicht benutzen.

OMNI ON, MONO

"Noch schlimmer", da soll eine Stimme alles spielen ???

OMNI OFF, POLY (MIDI MULTI MODE)

Wenn zusätzlich noch eine dynamische Stimmenzuordnung dazu kommt ist das ideal ! Man kann damit aus nur einem Gerät mehrere unterschiedliche Sounds rausholen und durch unterschiedlichen MIDI - Kanäle steuern lassen.

OMNI OFF, MONO

Dieser Modi "verheizt" MIDI Kanäle, denn das Gerät empfängt auf jeder Stimme nur von einem bestimmten MIDI Kanal.

5.2 Das MIDI Übertragungs Format

Generell bedeutet ein gesetztes 8. Bit, daß es sich um einen Befehl handelt ! Die Daten, die nach einem solchen Befehl kommen dürfen das 8. Bit nicht verwenden, können also nur in einem Bereich zwischen 0 - 127 liegen.

5.3.1 Spielen einer Note

Was hier kommt, sendet ein Keyboard beim Herunterdrücken einer Taste.

z.B. NOTE an

1.Byte, Status Note an (Bit 4-8), MIDI Kanal (Bit 1-4)

%10010000 ($90) + MIDI Kanal %0000 ($0) bis %1111 ($F)

2.Byte, Nummer der Note (7 Bit)

wobei $0 der tiefsten und $7F der höchsten Note entspricht

Beispiel einer Oktave:

Kontra	C	36	F#	42
	C#	37	G	43
	D	38	G#	44
	D#	39	A	45
	E	40	A#	46
	F	41	H	47

u.s.w.

3.Byte, Velocity (Anschlagdynamik - 7 Bit)

wobei $0 der tiefsten und $7f der höchsten Note entspricht

Wenn die eingeschaltete Note wieder ausgeschaltet werden soll, wird nochmals die gleiche Befehlsfolge geschickt. Das bedeutet, daß der Ton so lange eingeschaltet bleibt (gespielt wird) bis wieder ein Signal auf gleiche Tonhöhe geschickt wird.
Die meisten Keyboards senden beim Ausschalten der Note einen Velocity Wert von Null, es sind aber auch andere Werte möglich.

5.3.2. Pitch Bend Signale

Der Pitch Bend ist meist ein schwarzes Rad an dem Keyboard eines Synthesizers oder Samplers, beim Betätigen dieses Rades, wird die Note der gerade am Keyboard ("Klaviatur") gedrückten Taste je nach Einstellung etwas nach oben oder nach unten transponiert. Je nach Fähigkeiten und Einstellung des Tonerzeugers wird die Note um eine halbe Note (Maximaleinstelllung des Pitch Benders), eine Oktave oder möglicherweise noch mehr transponiert.

Die folgenden drei Bytes sendet ein Keyboard beim benutzen des Pitch Bend:

1. Byte, Pitch Bend

%11100000 ($E0) + MIDI Kanal ($0 bis $F)

2. Byte, niederwertige 7 Bit Pitch Bend Daten

meist unbenutzt

3. Byte, höherwertige 7 Bit Pitch Bend Daten

$7F -- höchste Frequenz
|
$40 -- Mittelstellung
|
$0 -- niedrigste Frequenz

7 Bit bedeutet, das eine Zahl > $7F (127) einen Übertrag erzeugt, der auf das nächste Byte (dem höherwertigen), in diesem Fall ins dritte Byte übertragen wird. Das wird uns bei MIDI-Übertragungen noch öfters begegnen, da das achte Bit ja für die Befehle reserviert ist, d.h. Datenbytes können nicht größer als 7 Bit werden.

5.3.3. Programm Change

Wechseln der Klänge auf den Geräten

1. Byte, Umschalten (Befehlsbyte + MIDI - Kanal)

%11000000 ($C0) + MIDI Kanal ($0 bis $F)

2. Byte, 7 Bit Klangnummer

$0 bis $7f

5.3.4. Aftertouch

Diese Funktion bietet eine zusätzliche Kontrolle des Tons beim Spielen eines Keyboards. Das heißt, nach dem Herunterdrücken einer Taste kann man zusätzlich durch mehr oder weniger heftiges Herunterdrücken oder Nachlassen dieser Taste (meist) ein Vibrato erzeugen. Nun, es gibt hier zwei Arten dieser Funktion:

a) Den monophonen Aftertouch, der auf das ganze Instrument und nicht nur auf die mit Druck versehene Taste wirkt:

Format:

1. Byte, monophoner Aftertouch

%11010000 ($D0) + MIDI Kanal ($0 bis $F)

2. Byte, Größe der Veränderung

$0 keine bis $7F volle Wirkung

b) Den polyphonen Aftertouch, bei dem jede Stimme einer gedrückten Taste durch entsprechenden Druck mit einem mehr oder weniger großen Wirkung eines Effekts (meist Vibrato) versehen werden kann.

1. Byte, polyphoner Aftertouch

%10100000 ($A0) + MIDI Kanal ($0 bis $F)

2. Byte, Note, auf die es wirkt

$0 bis $7F

3. Byte, Intensität

$0 keine bis $7F volle Wirkung

5.3.5 Control Changes

Ich kann hier nicht alle möglichen Control Change Befehle aufführen, da viele für den Spieleprogrammierer nicht so interessant sind und die Technik sich natürlich weiterentwickelt, d.h. viele Werte, die ich als unbenutzt kenne, sind möglicherweise von neueren Geräten schon belegt.

Bei den folgenden Parametern ist das 1. Byte immer

1. Byte, Controll Change + Midi Kanal

%10110000 ($80) + MIDI Kanal ($0 bis $F)

Übersicht:

$0 - $1F Regler , $20 - $3F reserviert , 40 - 5F Schalter
$60 - $79 reserviert , $7A - $7F Channel Mode Befehle

5.3.5.1 Modulations Rad

Die Modulation kann bei vielen Samplern u. Synthesizern sowohl auf Tonhöhe als auch auf Filterstellung wirken, bei einigen Geräten sogar auf das Mischungsverhältnis von Oszillatoren, Resonanz u.v.m.

Meist wird mit diesem Rad ein Oszillator gesteuert, der im Niederfrequenzbereich die Tonhöhe des Orginaltons ständig um seinen Originalwert von oben nach unten variiert.

2. Byte, $1 => Modulations Rad

3. Byte, höherwertige 7 Bit

$7F -- Volle Modulation
I
$40 -- Mittelstellung
I
$0 -- keine Modulation

5.3.5.2. Breath Controller

Der Anschluß für ein solches Gerät ist meist bei Yamaha Synthesizern zu finden, es ist ein Mundstück, mit dem man je nach Einstellung der Klangparameter, Tonhöhe und/oder Lautstärke mittels Hineinblasen beeinflussen kann.

2. Byte, $2 => Breath Controller

3. Byte, Intensität 0 (keine) bis $7F (voll)

5.3.5.3. Fuß Pedal

Eine Art Gas - Pedal für Synthesizer, je nach Gerät und Einstellungen kann man damit natürlich auch verschiedene Effekte erzeugen, wird jedoch meist zur Lautstärken-Regelung per Fuß genutzt.

2. Byte, $4 => Foot Pedal Controller

3 Byte, Intensität 0 (keine) bis $7F (voll)

5.3.5.4. Portamento Time

Wenn Portamento am empfangenden Gerät eingeschaltet ist, tritt beim Wechsel der Noten eine fließende Tonhöhenänderung auf. Mit der Portamento Time gibt man die Geschwindigkeit der Änderung an.

2. Byte, $5 => Portamento Time

3. Byte, Intensität $0 (keine) bis $7F (voll)

5.3.5.5. Data Entry

Regler am Synthesizer zur Dateneingabe, kann unterschiedlich belegt sein !

2. Byte, $6 => Data Entry

3. Byte, $0 bis $7F (je nach Funktion)

5.3.5.6. Gesamt Lautstärke

Lautstärke des gesamten Synthesizers

2. Byte, $7 => Main Volume

3. Byte, 0 (keine) bis $7F (volle Lautstärke)

5.3.5.6 Ein/Ausschalter

2. Byte,

$40 Hold/Sustain Pedal $41 Portamento
$42 Sustenuto Pedal $43 Soft Pedal

3. Byte, 0 = AUS - $7F =AN

5.3.5.7. +/- Tasten zum Ändern der Daten

Stufenweises Einstellen des DATA ENTRY !

2. Byte, $60 => +1 / $61 => -1

5.3.5.7 Channel Mode Befehle

Spezielle System-Einstellungen am Gerät:

2. Byte, $7A Local Control, 3. Byte, 0 = AUS - $7F = AN

"entkoppelt" Keyboard von Tonerzeuger, wenn Sie sich im gleichen Gerät befinden.

2. Byte, $7B All Notes Off   3. Byte, immer 0

Schaltet alle Noten aus, damit kann man Fehler bei der MIDI Übertragung (sogenannte MIDI-Hänger), z.B. wenn ein NOTE ON Signal aber kein NOTE OFF Signal gesendet wurde, ausbügeln.

2. Byte, $7C OMNI MODE OFF   3. Byte, immer 0

siehe 5.2.2.

2. Byte $7D OMNI MODE ON   3. Byte, immer 0

siehe 5.2.2.

2. Byte $7E MONO ON/POLY OFF   3. Byte, $0 bis $7F
Anzahl der MIDI Kanäle
siehe 5.2.2.

2. Byte $7F POLY ON/MONO OFF   3. Byte, immer 0

siehe 5.2.2.

5.3.6. Unter anderem gibt es noch die Common Commands

Von diesen Befehlen werden alle Geräte beeinflußt

1. Byte, %11110001 ($F1), %11110100 ($F4), %11110101 ($F5)

nicht definiert

1. Byte, $F2 Song Position
2. Byte, niederwertige 7 Bit der Posistion im Song eine Einheit = 6 Midi Clockimpuls und 24 Impulse ergibt 1/4 Note
3. Byte, höherwertigen 7 Bit der Position im Song

1. Byte, $F3 Song select
2. Byte, 7 Bit enthalten die Song Nummer

1. Byte, $F6 Tune request

1. Byte, $FF System Reset

5.3.7 Die Real Time Information

Signale, die in Echtzeit zu Syncronisation von Instrumenten, z.B. Rythmußgeräte, Bandmaschienen benötigt werden.

Immer nur ein Byte !

$F8 Timing Clock (24 Impulse = 1/4 Note)

$F9 Nicht definiert

$FA Start setzt den Zeiger für die Song-Position an den Anfang

$FB Continue

$FC STOP

$FD Nicht definiert

$FE Active sensing, eine Art Pausezeichen, wird den Geräten alle 300 ms gesendet, um sie bereit zu halten.

5.3.8. System Exclusive Daten

Die System Exclusive Daten sind von Synthesizer zu Synthesizer unterschiedlich, es gibt sie für alle möglichen und "unmöglichen" MIDI Geräte, sofern sie programmierbar sind. Leider gibt es nur sehr wenige einheitliche Formate bei den System Exclusive Daten,

z.B. der MIDI Sample Dump Standard. In den anderen Bereichen kocht jeder Hersteller eines programmierbaren Geräts meist sein eigenes Süppchen, was es für den Spieleprogrammierer unmöglich macht, Sounds für seine Programme mit zu übertragen, außer er ist sich sicher, daß genau das Gerät angeschlossen ist, für das er die Sounds überträgt (z.B. MT32 oder CZ XXXX bei einigen Adventures).

$F0 zeigt an, daß jetzt System Exclusive Daten kommen
$F7 das keine real Time Daten übertragen werden
ein Byte für den Hersteller z.B. $41 ROLAND, $23 YAMAHA
$0 - $F MIDI Kanal, auf dem die kommenden Daten empfangen werden

.

.

Die dazwischen liegenden Bytes sind den Herstellern entsprechender Geräte überlassen, Informationen dazu sind entweder in den Anleitungen der Geräte oder eben bei dessen Herstellern erhältlich.

.

.

$F7 zeigt das Ende der Übertragung von exclusiv Daten an

5.3.9 Der MIDI Sample Dump Standard

Gerade für Benutzer eines Falcon dürfte dieser Standard in Sachen Klangdatenübertragung sehr interessant sein, läßt sich doch auf diesem Weg ein komplettes Sample von jedem beliebigen Sampler, für viele gibt es große Soundbibliotheken, in den Falcon, und das ohne Qualitätseinbußen, die durch direktes Absamplen am Audio Ausgang des Samplers in Kauf genommen werden müssen, in den Falcon übernehmen.

Es gibt im Prinzip zwei Möglichkeiten, die Daten zu übertragen, zum einem im Handshake, d.h. daß der Computer und der Sampler über zwei MIDI Kabel miteinander verbunden sind. Man gibt dem Sampler die Anweisung, den Sample zu senden und der Computer verarbeitet dann die Daten, wobei nur ein MIDI Kabel nötig ist.

Im Handshake sieht die Übertragung in etwa so aus:

Computer "fragt"

$F0 => Sytemexklusiv , $7E => kein realtime dann ein Byte MIDI Kanal

2 Byte mit niederwertigem und dann höherwertigen 7 Bit der Samplenummer

$F7 Ende der Übertragung

Sampler "antwortet" mit 21 Bytes langem Sample Header

$F0, $7E

ein Byte MIDI Kanal

2 Byte mit niederwertigem und dann höherwertigen 7 Bit der Samplenummer

1 Byte, Format des Sample, 8 bis 28 Bit

3 Bytes mit der Periode des Samples in nano $=10^{-9}$ s natürlich im üblichen 7 Bit pro Byte, mit dem niederwertigen Byte zuerst.

3 Byte für die Länge des Samples (7 Bit pro Byte)

3 Bytes Loop Start Punkt (7 Bit pro Byte)

3 Bytes Loop Endpunkt (7 Bit pro Byte)

1 Byte Looprichtung, 0 = vorwärts / 1 = rückwärts

$F7 Ende ...

Computer antwortet:

$F0, $7E, MIDI Kanal, $7F

1 Byte Paketnummer (7 Bit), wenn die Nummer > 127 ist, fängt diese wieder mit 0 an. Bei einem Fehler bei der Übertragung der folgenden 120 Daten (7 Bit) Bytes mit Sampledaten kann man mit Hilfe dieses Datenbytes das Paket nochmals anfordern.

$F7 Ende...

Sampler sendet:

$F0, $7E, MIDI Kanal, $02, Paketnummer

120 Byte Sampledaten = ein Paket, dessen Format etwa so aussieht:

z.B. 12 Bit

```
   MID        LOW
%0nnnnnnn  %0nnnnn00
```

oder 16 Bit

```
  HIGH       MID        LOW
%0nnnnnnn  %0nnnnnnn  %0nn00000
```

1 Prüfbyte, XOR aller Daten zwischen $F0 und diesem Byte

$F7 Ende...

Computer gibt darauf entweder:

Acknowledge, wenn alle Daten stimmen

$F0, $7E, MIDI Kanal, $7F, nächste Paketnummer, $F7

oder

$F0, $7E, MIDI Kanal, $7F, letzte Paketnummer, $F7

wenn die Daten nicht korrekt übertragen wurden.

usw.

Wenn man die Übertragung vorzeitig beenden möchte, sendet man

Cancel,

$F0, $7E, MIDI Kanal, $7D, Paketnummer, $F7

Wenn Sampler und Computer nur mit einem MIDI Kabel verbunden sind (also kein Handshake erfolgt), wartet der Sampler zwei Sekunden auf eine Antwort und sendet dann weiter !

Das wars !

B.6. Der Aufbau wichtiger Pixelgrafik-Formate

Für den Spieleprogrammierer ist das Importieren von Grafik eine der wichtigsten Aufgaben überhaupt, denn kaum eine Gattung von Spielen kommt so ganz ohne Grafik aus. Nun, es ist für den Hobby - Programmierer nicht immer ganz so einfach, die mit einem Standard-Zeichenprogramm entwickelten Bilder in das eigene Programm einzubinden, denn diese sind meist gepackt, enthalten einen Header mit Daten zu Format, Palette u.v.m. . Aus diesem Grund möchte ich hier die für den Spieleprogrammierung wichtigsten Bildformate beschreiben.

6.1 Komprimierung bzw. Dekomprimierung von Bilddaten

Packen u. Entpacken von Daten ist ein komplexer Themenbereich, daher möchte ich hier dem Leser einen kurzen Überblick verschaffen, um dann bei den Bildformaten genauer auf die einzelnen Verfahren eingehen zu können. Die hier besprochenen Packverfahren sind allesamt verlustfreie Packverfahren, d.h. "solche Hämmer wie 'JPEG' habe ich mir erspart". Verlustfrei bedeutet, daß man nach dem Auspacken genau das Gleiche erhält wie das, was man eingepackt hat.

Nun, was heißt überhaubt 'packen' ?

Packen ist ein Verfahren um die Menge der anfallenden Daten zu reduzieren. Gerade bei Bildern ist das Packen eine effiziente Methode um den Massenspeicher ein klein wenig zu entlasten. So lassen sich Bilder meist bis auf weniger als die Hälfte ihrer Länge komprimieren. Das liegt an den vielen gleichfarbigen Teilen oder oft vorkommenden Mustern aus denen ein Bild besteht, solche gleiche oder ähnliche Datenstrukturen lassen sich extrem gut zusammenpacken. Dazu wird ein Algorithmus entworfen, der die Bilder codiert und dabei die gleichen oder ähnlichen Daten in einer Struktur zusammenfaßt.

Es gibt unzählige Verfahren, Daten zu packen, die teilweise stark an die zu packenden Daten angepaßt wurden. Hier zwei der gebräuchlichsten.

6.1.1 Run length Compression bzw. Run Lengt Encoding (RLE)

'Run length Compressien' wird in vielen Bild-Formaten (mit)verwendet, z.B. IFF, Degas Elite (PC1-PC3), IMG, STAD (.PAC), Spectrum 512 (SPC), PCX, MSP u.v.m.

Die einfachste Art der RLE ist, einfach die Menge der gleichen Punkte durch z.B. 1 Byte Länge und 1 Byte Farbe zu codieren. Allerdings ist das wenig effizient, da an vielen Stellen im Bild mehrere andersfarbige Pixel hintereinander kommen, daher wird RLE meist in leicht abgewandelter Form oder mit anderen Verfahren zusammen verwendet.

6.1.2 Das Lempe-Ziv Welsh Verfahren (LZW)

Das LZW-Verfahren basiert auf dem Prinzip, das nicht Zeichen sondern Kompression (LZW) Codes übertragen werden. Die zu übertragenen Zeichen werden in einer String Tabelle Codes zugeordet. Dieses Verfahren ist sehr leistungsfähig und wird unter anderem für Compuserves ".GIF "Format verwendet.

6.2. Aufbau der File Formate

6.2.1 Neochrome

Header (Formatkopf)

Byte:

$0 - $03	nicht definiert
$4 - $23	16 Worte Palette, also genau so wie es 'setpalette' (XBIOS 4) akzeptiert, es ist auf dem STE/TT/Falcon möglich, das niederwertige Bit zu setzen und somit aus 4096 Farben auszuwählen, obwohl die mir bekannten NEOCHROME - Versionen das Auswählen aus so vielen Farben nicht unterstützen, stellen sie alle 4096 Farben dar, da die 16 Worte wohl unmaskiert in die Farbregister übertragen werden.
$24 - $30	nicht definiert.
$31	höherwertiges Nibble (4 Bit) Startfarbe der Palettenanimation Nibble (4 Bit) Endfarbe der Palettenanimation
$32 - $34	Geschwindigkeit der Palettenanimation, angegeben in der Anzahl der aufgebauten Monitorbilder (VBL's) 50/60 Hz zwischen den Änderungen der Palette.
$35 - $7F	nicht definiert
32000 Byte	Bildschirm

Danach kommt immer ein 16 Farben 320*200 Bild und zwar mit dem Aufbau des Bildschrimspeichers in der niedrigen Auflösung.

6.2.2 Degas Elite

Byte:

$0 - $1	Auflösung: 0=niedrig; 1=mittel; 2=hoch ist Bit 15 gesetzt, sind die Bilddaten komprimiert
$2 - $22	16 Worte Farbpalette

dann folgen die Bilddaten, wenn unkomprimiert 32000 Bytes im Bildschirmspeicher Format

gefolgt von

4 Worten linke Begrenzungen für die 4 Farbanimationskanäle
4 Worten rechte Begrenzungen für die 4 Farbanimationskanäle
4 Worten Richtungsflag für die 4 Farbanimationskanäle
4 Worten Zeitverzögerung für die 4 Farbanimationskanäle

Komprimierung RLE (siehe auch Targa und IFF Format)

Dekomprimierung: (einer Bitplane)

1. Länge d. Bilddaten = Länge d. Bilddaten -1, falls Länge 0 ENDE
2. Nehme Byte aus Bilddaten
3. Falls Signed Byte >=0 gehe zu 7.
4. Falls Byte =$80 gehe zu 1.
5. copiere Byte * nächstes Byte aus Bildaten in den Bildspeicher
6. gehe zu 1.
7. lösche Bit 7 in Byte
8. Länge der Bilddaten = Länge d. Bilddaten -1
10. Kopiere Byte kommenden Bytes in den Bildschimrspeicher
11. gehe zu 1.

Bei Farbauflösungen (320*200*4 und 640*200*2) wird Zeile für Zeile decomprimiert. Wobei die Bitplanes wie beim IFF-Format zeilenweise hintereinander angeordnet sind und auch genauso komprimiert sind.

6.2.3 TARGA

Das Targa Format ist extrem flexibel und wird hauptsächlich bei True Color Bildern verwendet.

Achtung ! die hier angegebenen Wörter sind im "Little Endian Format" angegeben, d.h. in einem Wort befindet sich erst das nieder- dann höherwetiges Byte. Woran man merkt, daß dieses Format aus der PC - Welt stammt.

Byte:

$0 - $11 Header

$0	ident_len, Länge eines Kommentar Blocks
$1	cmap_type, wenn 1 Palette sonst RGB
$2 - $3	cmap_origin, Lage von color 0 in Palette
$4 - $5	cmap_len, Anzahl der Farben in Palette z.B. 256
$6	centry_size, Bittiefe der Palette
$7 - $8	image_xorg, X - Wert von links unten
$9 - $10	image_yorg, Y - Wert von links unten
$11 - $12	image_width, Breite des Bildes in Punkten
$13 - $14	image_height, Höhe des Bildes in Punkten
$15	pixel_size, Anzahl der Bits pro Pixel 16, 24, 32 Bits
$17	
$18 - $19	image_discr, weitere Informationen zum Bildaufbau, normalerweise 0

sonst

Bits 0 - 3	Attribut Bits pro Pixel (Füllbits)
Bit 4	immer 0
Bit 5	0 = Ursprung in der linken unteren Ecke. Aufbau des Bildes von links nach rechtsund von unten nach oben 1 = Ursprung in der linken oberen Ecke. Aufbau des Bildes von links nach rechts und von oben nach unten

Bit 6 - 7 00 Die Zeilen folgen nacheinander
01 Erst die geraden dann die ungeraden Zeilen
10 Die Bildzeilen werden in vier Blöcke unterteilt, {0, 4, 8, ...} {1, 5, 9,...} {2, 6, 10, ...} {3, 7, 11, ...}
11 Reserviert

Kommentar Block, bis zu 255 Bytes lang

Palette, falls erforderlich, in folgendem Format:

bei 32 Bit 4 Byte pro Eintrag "BGRX"
bei 24 Bit 3 Byte pro Eintrag "BGR"
bei 16 Bit 2 Byte pro Eintrag, %Xrrrrrgg gggbbbbb (Intel Format)

Bildinformationen:

Der Aufbau der Farbpalette entspricht bei einem Bild ohne Palette (direct color) dem Pixelformat.

Komprimierung RLE

Byte = %innnnnnn

%nnnnnnn = menge-1

i = ID Bit, wenn gesetzt komprimiert, sonst folgen %nnnnnnn+1 Bytes unkomprimierte Daten

Wenn also das ID Bit gesetzt ist wird die folgende Information über die Pixelfarbe oder Farbnummmer in der Palette %nnnnnnn+1 mal auf den Bildschirm gebracht. Wenn nicht, werden die nächten %nnnnnnn+1 Pixelfarbwerte oder Farbnummern in der Palette auf den Bildschirm gebracht.

6.2.4 IFF (Interchange File Format)

In diesem Format findet man auf dem AMIGA und gelegentlich auch auf dem PC Bilder, Animationen und Samples. Auf dem ST wird dieses Format auch von einigen Programmen, wie z.B. Deluxe Paint 2 unterstüzt. Dieses Format wurde von Electronic Arts für die damals neuen AMIGA entwickelt und hat sich diesem Gerät schnell zum Standard entwickelt.

Das Konzept, das hinter den IFF Dateien steht, ist, ein leicht erweiterbares Dateiformat zu sein, das für alle möglichen Datentypen verwendet werden kann. Das hat man durch die sogenannten Chunks gelöst.

Ein Chunk ist ein Datenblock, dieser Block kann alle mögliche Datentypen Enthalten, welcher üblicherweise mit einer Identifikation (ID) beginnt.

So ein ID ist ein Reihe von 4 ASCII Zeichen, also genau einem Langwort (was es dem Maschinenprogrammierer leichter macht), das die einzelnen Blöcke (Chunks) einer IFF Datei kennzeichnet und voneinander abtrennt.

Nach der ID folgt ein Langwort mit der Länge der Daten im Chunk und dann die Daten.

Wenn man also eine IFF Datei lesen möchte, muß man sich von Chunk zu Chunk hangeln, in dem man die ID eines Chunks prüft und, falls es sich nicht die gewünschten Daten handelt, den Datenblock überspringt und das nächste ID testet. Denn mehrere Bild-, Sample-, Text- etc. Blöcke können sich in einer einzigen Datei befinden.

Die üblichen Chunks einer IFF Bilddatei:

ID

"FORM" leitet ein Objekt ein (z.B. Bilddatei), gefolgt von einem Langwort mit der Länge des Objekts und ein ID für die Art des Objekts

ILBM	=	InterLeaved BitMap (Bilddatei)
8SVX	=	8 Bit Sampled Voice (Sample)
ANBM	=	ANimated Bitmap
GSCR	=	General use musical SCore
PICS	=	MacintoSh PICture
USCR	=	Uhuru sound software musical ScoRe
UVOX	=	Uhuru sound software Macintos VOice
SMUS	=	Simple MUSic Score
VDEO	=	deluxe video Construction Srt ViDEO

u.v.m

"BMHD" BitMap HeaDer (Bildformat)

1 Langwort - Länge des Chunks
1 Wort Breite des Bitmapausschnitts
1 Wort Höhe des Bitmapausschnitts
1 Wort X Position der Bitmap auf der Grafikseite (meist 0)
1 Wort Y Position der Bitmap auf der Grafikseite (meist 0)
1 Byte Menge der Bitplanes
1 Byte Maske, 0 = keine Maske
1 = Maske als Plane im Body Chunk
2 = Bild durchsichtig wo transparent Color
3 = alle Pixel außerhalb des Umrisses transparent
1 Byte komprimiert, 1 = komprimiert
1 Byte reserviert
1 Byte transparent Color
1 Byte Pixel ratio x, Pixelgrößenverhältnis X
1 Byte Pixel ratio y, zu Pixelgrößenverhältnis Y
1 Wort Breite der Grafikseite
1 Wort Höhe der Grafikseite

"CMAP" ColorMAP (Palette)

1 Langwort - Länge des Chunks

Die Palettedaten werden immer auf 8 Bit pro Farbe ausgelegt. Das bedeutet, daß sich die 4096 Farben des STE alle im höherwertigen Nibble (4 Bit) der jeweiligen Farbkomponente.

z.B. eine STE Palettenregister mit

rot = 12, grün = 5, blau = 1

erzeugt folgende 3 Bytes eines IFF Paletteneintrags

R G B

1. Byte = 192, 2. Byte = 80, 3. Byte = 16

"BODY"

1 Langwort - Länge des Chunks

Entpacken von IFF Planes und Zeilen:

Das Entpacken funktioniert wie beim Targa Format, allerdings sind die Bilddaten bei IFF planeweise und nicht pixelweise wie im Targa Format organisiert.

Aufbau einer Zeile:

1. Zeile - Plane 1
1. Zeile - Plane 2
....
....
1. Zeile - Plane n

B.7. Nutzen des DSP 56001 bei der Spieleprogrammierung

Leider hatte ich den F030 nur etwa 3 Wochen zu Verfügung, einer jener Modelle, die von ATARI einigen Händlern als Vorführgerät geliefert wurde. Exakt das Modell, das auch in einigen Zeitungen von innen zu bewundern war (ST Computer, ATARI Journal etc.). Ich konnte mich daher nicht allzu lange mit dem DSP beschäftigen, auch weil der Falcon noch andere faszinierende Geheimnisse verbarg, die erforscht werden mußten.

Ich kann hier nur einen kurzen Überblick über die Programmierung des DSP bieten, somit ist dieses Kapitel auch mehr zum Nachschlagen gedacht, weitere Informationen sind dem "DSP 56000/1 Users Manual" von MOTOROLA zu entnehmen, mehr würde den Rahmen des Buches sprengen und wäre höchstens eine Idee für ein weiteres Buch.

7.1 Was ist ein Signalprozessor ?

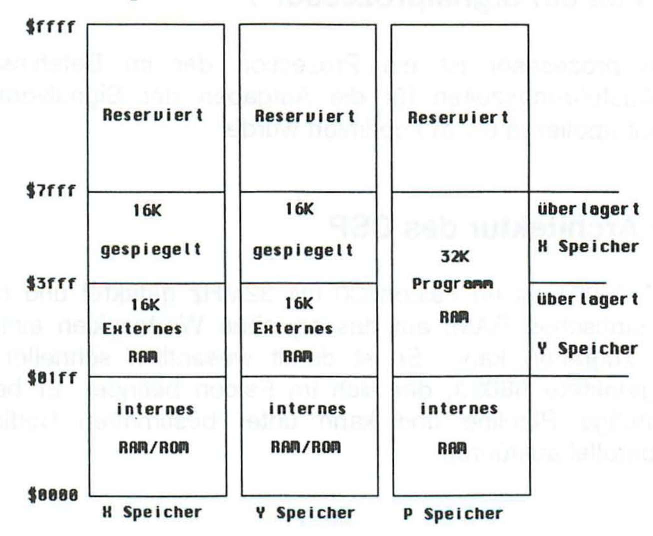
Ein Signalprozesssor ist ein Prozessor, der im Befehlssatz und seinen Ausführungszeiten für die Aufgaben der Signalverarbeitung (Filtern, Interpolieren u.v.m.) optimiert wurde.

7.2 Die Architektur des DSP

Der DSP 56001 ist im Falcon030 mit 32MHz getaktet und hat 96kB eigenes statisches RAM, auf das er, ohne Wartezyklen einlegen zu müssen, zugreifen kann. Er ist damit wesentlich schneller als ein 16MHz getaktete 68030, der sich im Falcon befindet. Er besitzt eine dreistufige Pipeline und kann unter bestimmten Bedingungen Befehle parallel ausführen.

7.2.1 Das Buskonzept

Prinzipiell ist der DSP schon ein Computer in sich, mit Schnittstellen, Bus, RAM und ROM. Im Gegensatz zur 68000er Familie besitzt der DSP Harvard-Architektur, so sind Daten und Programmspeicher getrennt. Der DSP kennt drei Arten von Speicher, X Datenspeicher, Y Datenspeicher und P Programmspeicher. Die ersten 512*24 Bit von X und Y Datenspeicher sind jeweils ROM und die folgenden 512*24Bit Bytes sind jeweils RAM. Erweitern läßt sich der X- als auch der Y-Speicher, extern bis aus 64k*24 Bit. Der Programmspeicher des DSP 56001 besteht aus 512*24 Bit RAM Speicher; dieser läßt sich ebenfalls extern auf 64k*24 Bit erweitern. Im Falcon030 hat ATARI den DSP mit 32K*24 Bit externem RAM erweitert. ATARI benutzt das externe 32k*24 Bit RAM sowohl für X- und Y- also auch für P- RAM, so daß diese sich überschneiden. Somit ist der Zugriff auf externes RAM im DSP, zumindest teilweise, wieder wie bei den Rechnern mit von Neumann Architektur (Daten und Programm in einem Speicher).



7.2.2 Die Interruptvektoren des DSP

In den ersten 64*24 Bit des DSP-Programmspeichers befinden sich die Interrupt Vektoren.

Beginnend an der Adresse P:$0000 oder P:$E000

Offset	IPL	Interrupt Geber
0	3	Hardware Reset
2	3	Stack Error
4	3	Trace
6	3	SWI (Software Interrupt) ähnl. TRAP beim 68000
8	0-2	IRQA Externer Interrupt
A	0-2	IRQB Externer Interrupt
C	0-2	SSI empfängt Daten
E	0-2	SSI empfängt Daten (Exception Status)
10	0-2	SSI sendet Daten
12	0-2	SCI sendet Daten (Exception Status)
14	0-2	SCI empfängt Daten
16	0-2	SCI empfängt Daten (Exception Status)
18	0-2	SCI sendet Daten
1A	0-2	SCI wartet
1C	0-2	SCI Timer
1E	3	NMI (nicht maskierbarer Interrupt) extern
20	0-2	Host empfängt Daten
22	0-2	Host sendet Daten
24	0-2	Host Befehl (voreingestellt)
26	0-2	als Host Befehl möglich
28	0-2	als Host Befehl möglich
2A	0-2	als Host Befehl möglich
2C	0-2	als Host Befehl möglich
2E	0-2	als Host Befehl möglich
30	0-2	als Host Befehl möglich
32	0-2	als Host Befehl möglich
34	0-2	als Host Befehl möglich
36	0-2	als Host Befehl möglich
38	0-2	als Host Befehl möglich
3A	0-2	als Host Befehl möglich
3C	0-2	als Host Befehl möglich
3E	0-2	Illegaler Befehl

Achtung ! der SCI-Port und die SCI-Register sollen laut ATARI bei DSP Software auf dem F030 nicht verwendet werden.

7.2.3 Die I/O Register des DSP

Die Adressen sind wie beim DSP üblich wortweise, also in 24 Bit-Schritten angegeben.

X:$FFC0 bis X:$FFDF Reserved

X:$FFE0	Port B - Bus Control Register (PBC)
X:$FFE1	Port C - Control Register (PCC)
X:$FFE2	Port B - Data Direction Register (PBDDR)
X:$FFE3	Port C - Data Direction Register (PCDDR)
X:$FFE4	Port B - Data Register (PBD)
X:$FFE5	Port C - Data Register (PCD)
X:$FFE6	Reserved
X:$FFE7	Reserved
X:$FFE8	Host Control Register (HCR)
X:$FFE9	Host Status Register (HSR)
X:$FFEA	Unbenutzt
X:$FFEB	Host Receive/Transmit Register (HRX/HTX)
X:$FFEC	SSI Control Register A (CRA)
X:$FFED	SSI Control Register B (CRB
X:$FFEE	SSI Status/Time Slot Register (SR/TSR)
X:$FFEF	SCI Receive/Transmit Register (RX/TX)
X:$FFF0	SCI Interface Control Register (SCR)
X:$FFF1	SCI Interface Status Register (SSR)
X:$FFF2	SCI Control Register (SCCR)
X:$FFF3	SCI Transmit Data Address Register (STXA)
X:$FFF4	SCI LOW-REC/XMIT Data Register (SRX/STX)
X:$FFF5	SCI MID-REC/XMIT Data Register (SRX/STX)
X:$FFF6	SCI HI-REC/XMIT Data Register (SRX/STX)
X:$FFF7	Reserved
X:$FFF8	Reserved
X:$FFF9	Reserved
X:$FFFA	Reserved
X:$FFFB	Reserved
X:$FFFC	Reserved
X:$FFFD	Reserved
X:$FFFE	Port A - BUS Control Register (BCR)
X:$FFFF	Interrupt Priority Register (IPR)

7.3. Programmierung des DSP

7.3.1 Die Register des DSP

Register zur Adressierung

Die Adress-Erzeugungs-Einheit (AGU) des DSP besteht aus zwei Arithmetisch Logischen Einheiten (ALU). Jede ALU hat ihre ihr eigens zugeordneten Register und kann bei Verwendung dieser Register parallel zur zweiten ALU 16 Bit Adressberechnungen vornehmen. Es ist aber auch möglich, für eine ALU die Register der anderen mitzubenutzen, was dann eine Parallelverarbeitung beider ALU's ausschließt.

LOW ADDRESS ALU

R0-R3, 16 Bit Adressregister N0-N3, 16 Bit Offset Register
M0-M3, 16 Bit Modifizier Register

HIGH ADDRESS ALU

R0-R7, 16 Bit Adressregister
N0-N3, 16 Bit Offset Register
M0-M3, 16 Bit Modifizier Register

Datenregister

X , 56 Bit bestehend aus

X0&X1, 24 Bit Datenregister

Y , 56 Bit bestehend aus

Y0&Y1, 24 Bit Datenregister

Akkumulatoren

A , 56 Bit bestehend aus

A0, 24 Bit Akku
A1, 24 Bit Akku
A2, 8 Bit Akku

B, 56 Bit bestehend aus

B0, 24 Bit Akku
B1, 24 Bit Akku
B2, 8 Bit Akku

Programm Kontroll Register

LA, LOOP Adresse bzw. Adresse des letzten Befehls im LOOP
LC, LOOP Zähler - Menge der Wiederholungen (auch für REP)

OMR, 8 Bit Operating Mode Register

SR, Status Register, bestehend aus CCR - 8 Bit und MR - 8 Bit

SP, 6 Bit Stackpointer

System Stack, bestehend aus 15*32 Bit internem Speicher aufgeteilt in SSH und SSL je 16 Bit breit.

7.3.2 Die Adressierungsarten des DSP

Glücklicherweise ist der Syntax des DSP-Assemblers dem der 68000er Familie in vielem ähnlich.

Wordbreite des DSP:

8 Bit = Byte
16 Bit = Short Byte
24 Bit = Word
47 Bit = Long Word
56 Bit = Accumulator

S = Zugriff auf den System Stack
C = Zugriff auf das Programm Controller Register
D = Zugriff auf das Data ALU Register
A = Zugriff auf Address ALU Register
P = Zugriff auf den Programm Speicher
X = Zugriff auf den-X Speicher
Y = Zugriff auf den-Y Speicher
L = Zugriff auf den-L Speicher (48 Bit von/zu X: und Y:)
XY = Zugriff auf X&Y Speicher (24 Bit je von/zu X: und Y:)

Register direkt

a) Daten oder Kontroll Register direkt (nur C, D)

```
MOVE X1,LC  ; Kopiere X1 nach LC
```

b) Adress Register direkt (nur A)

```
MOVE R1,R2 ; kopiere R1 nach R2
```

c) Adress, Offset Register direkt (nur A)

```
MOVE R1,N1  ; kopiere R1 nach N1
```

d) Adress, Modifizier Register direkt (nur A)

```
MOVE R2,M3  ; kopiere R2 nach m3
```

Adress Register Indirekt

e) Adress Register indirekt

möglich für P, X, Y, L, XY

```
MOVE  A1,X:(R0)  ; Inhalt von A1 in Adresse worauf R0 zeigt
```

f) Adress Register indirekt mit postincrement (+1, danach)

möglich für P, X, Y, L, XY

```
MOVE B0,Y:(R1)+  ; B0 an Adresse R1 dann R1=R1+1
```

g) Adress Register indirekt mit postdecrement (-1, danach)

möglich für P, X, Y, L, XY

```
MOVE Y0,Y:(R3)-       ; Y0 in Adresse R3 dann R3=R3-1
```

h) Adress Register indirekt mit postincrement Offset Register

möglich für P, X, Y, L, XY

```
MOVE X1,X:(R2)+N2   ; X1 in Adresse R2
                    ; dann R2=R2+N2
```

i) Adress Register indirekt mit postdecrement Offset Register

möglich für P, X, Y, L

```
MOVE X:(R4)-N4,A0     ; Inhalt von Adresse R4 in A0
                      ; dann R4=R4-N4
```

j) Adress Register indirekt mit Offset Register als Index

möglich für P, X, Y, L

```
MOVE Y1,X:(R6+N6)  ; Inhalt von Y1 in Adresse (R6+N6)
```

k) Adress Register indirekt predekrement (-1, davor)

möglich für P, X, Y, L

```
MOVE X:-(R5),B1  ; R5=R5-1, Inhalt von R5 in B1
```

SPECIAL

l) Konstanten Adressierung (nur in P:)

```
MOVE #$123456,A0  ; Konstante in 24 Bit Register
MOVE #$123456,A   ; Konstante in 56 Bit Register
MOVE #$801234,A   ; negative Konstante in 56 Bit Register
```

m) Konstanten Adressierung - Short Byte (nur in P:)

```
MOVE #$FF,A1  ; Konstante in z.B. A1,A2,B0,B2,Rn,Nm
MOVE #$1F,Y1  ; positive Konstante in z.B. X0,X1,Y0,Y1,A,B
MOVE #$1F,A   ; positive Konstante in z.B. X,Y,A,B
MOVE #$83,B   ; negative Konstante in z.B. X0,X1,Y0,Y1,A,B
```

n) Absolute Adressierung (nur in P:)

```
MOVE Y:$5432,B0  ; Inhalt von $5432 in B0
```

o) Absolute Adressierung - Short Address (in P:, X:, L:)

Die Adresse des Operanden darf nicht größer als 6 Bit sein. Die Adresse wird mit Nullen auf 16 Bit erweitert, wenn ein Operand oder Programmspeicher adressiert wird.

```
MOVE P:$3200,X0
MOVE A1,X:$3
```

p) Short Jump Address, springe zu mit 12 Bit angegebener Adresse

```
JMP $123    ; Adresse muß kleiner sein als $1000
```

q) I/O Adressierung - short Address

```
MOVEP A1,X:<<$FFFE ; Ziel darf hier nur von $FFC0 - $FFFF gehen
```

r) Implicit Reference, ändern eines nicht angegebenen Registers

Ein gutes Beispiel ist der 'REP' Befehl, der den LC ändert.

7.3.3 Übersicht der Befehle

Ich kann hier nur eine kurze Übersicht bieten, eine ausführliche Erklärung findet man im USER Manual von Motorola, wobei allein die Erläuterung des Befehlssatzes über 250 Seiten einnimmt.

Die Flags

Die E, U, N, Z, V, und C Bits sind echte condition codes, sie werden je nach Ergebnis einer Operation der ALU gesetzt. Bei Adress-Berechnungen oder Datentransfer werden sie nicht verändert. Das L Bit (latching overflow bit) wird gesetzt, wenn ein Überlauf im Accumulator auftritt (wie auch das V Bit) oder beim Wandeln von 56 Bit auf 24 Bit durch den Limiter.

*	wird gesetzt, je nach Ergebnis der ALU
-	wird nicht verändert
?	wird nur in bestimmten Situationen gesetzt
0	Das V Bit wird gelöscht

Befehl	Flags	Notiz	Funktion
	L EUNZVC		
ABS	* * * * * * _		Absoluter Wert
ADC	* * * * * * *		Addiere 48 Bit mit Übertr.
ADD	* * * * * * *	1	Addiere 24 Bit
ADDL	* * * * * ? *		Quelle + 2*Ziel
ADDR	* * * * * * *		Quelle + Ziel/2
AND	* _ _??0 _	8,9	logisches UND 24 Bit
ANDI	???????	2	UND 8 Bit konst. mit Reg
ASL	* * * * * ??	1,3	Accu nach links versch.
ASR	* * * * * 0?	4	Accu nach rechts versch.
BCHG	______?	5	teste u. ändere Bit
BCLR	______?	5	teste u. lösche Bit
BSET	______?	5	teste u. setze Bit
BTST	______?	5	teste Bit
CLR	* * * * * 0_		lösche Akku
CMP	* * * * * * *		vergleiche
CMPM	* * * * * * *		vergl. ABS(Quelle),Ziel
DIV	* ____ ? ?	8,9	Divid. Ziel=quot.& rest
DO	*______		Starte Hardw. Loop
ENDDO	_______		Beende Loop
EOR	*__??0_		Exklusive Oder 24 Bit
ILLEGAL			illegaler Befehl

Jcc	_______		springe wenn cc
JCLR	_______		springe wenn Bit gelöscht
JMP	_______		unbedingter Sprung
JScc	_______		springe Unterpr. wenn cc
JSCLR	_______		springe sub. wenn cc gel.
JSET	_______		springe wenn Bit gesetzt
JSR	_______		springe zu Unterprogramm
JSSET	_______		springe Unterpr. wenn Bit
LSL	* __??0?	8,9,10	schiebe nach links 24 Bit
LSR	* __??0?	8,9,11	schiebe nach rechts 24 Bit
LUA	_______		Load Updated Address
MAC	* * * * * *_		Add. Multi. zu Accu.
MACR	* * * * * *-		Add ger. Multi zu Accu.
MOVE	______ *		kopiere Quelle nach Ziel
MOVEC	???????	13	kopiere CR
MOVEM	???????	13	kopiere in P: oder von P:
MOVEP	???????	13	kopiere von/zu I/O mem.
MPY	* * * * * *_		vorz. ger. Multi.
MPYR	* * * * * *_		runde Vorz. ger. Multi.
NEG	* * * * * *_		negiere Accu.
NOP	_______		No Operation
NORM	*****?_	1	Normalize Accu. Iteration
NOT	* __??0_	8,9	%1 -> %0 & %0 -> &1
OR	* __??0_	8,9	ODER
ORI	???????	6	ODER 8 Bit konst. mit Reg.
REP	*_______		wiederh. nächst. Befehl
RESET	_______		RESET DSP
RND	* * * * * *_		runde Akku.
ROL	* __??0?	8,9,10	ROtate Left 24 Bit
ROR	* __??0?	8,9,11	ROtate Right 24 Bit
RTI	???????	12	ReTurn from Interrupt
RTS	* _______		ReTurn from Subroutine
SBC	* * * * * * *		Subtrah. 56 Bit mit Übert.
STOP	_______		stoppt den Prozessor
SUB	* * * * * * *	1	subtrahiere 24 Bit
SUBL	* * * * *? *		Quelle-2*Ziel
SUBR	* * * * * * *		Quelle-Ziel/2
SWI	_______		Software Interrupt

Tcc	_______	Übertrage wenn cc
TFR	* ______	Übertrage Akku. Akku
TST	* * * * * * *	vergleiche Akku.s
WAIT	_____	warte auf Interrupt

Notiz:

1 V wird gesetzt, wenn ein Überlauf bei einem 56 Bit Ergebnis. Es wird auch gesetzt, wenn das MS Bit des Ziels wegen eines shift left (z.B. ADDL, SUBL) geändert wurde. Ansonsten wird das V Bit gelöscht.
2 ? wird gelöscht, falls das Ziel das CCR ist, wenn das korrespondierende Bit in der Konstante gelöscht ist.
3 C wird gesetzt, wenn das Bit 55 der Quelle gesetzt ist. Sonst wird es gelöscht.
4 C wird gesetzt, wenn das Bit 0 in der Quelle gesetzt ist. Ansonsten gelöscht
5 C wird gesetzt, wenn das Bit #n in der Quelle gesetzt ist. Sonst gelöscht,
6 ? Wird gesetzt, falls das Ziel das CCR ist, wenn das correspondierende Bit der Konstante gesetzt ist. Ansonsten bleibt das Bit wie es ist.
7 C wird gesetzt, wenn Bit 55 des Ergebnisses gelöscht ist. Ansonsten wird es gelöscht.
8 N wird gesetzt, wenn Bit 47 des Ergebnisses gesetzt ist. Sonst wird es gelöscht.
9 Z wird gesetzt, wenn die Bits 24 - 47 Null sind. Sonst wir es gelöscht.
10 C wird gesetzt, wenn Bit 47 der Quelle gesetzt ist. Ansonsten gelöscht.
11 C wird gesetzt, wenn Bit 24 der Quelle gesetzt ist. Sonst gelöscht.
12 ? wird je nach Wert, der vom Stack geholt wurde, gesetzt.
13 ? wird gesetzt nach den korrespondierenden Bits der Quelle, wenn das Status Register (SR) das Ziel ist. Wenn SR nicht das Ziel ist , wird das L Bit, wenn ein Wert abgeschnitten wurde, gesetzt. Ansonsten werden die ? Bits nicht verändert.

CONDITON CODE	BEDEUTUNG	CCR
CC	Carry Clear	C=0
CS	Carry Set	C=1
EC	Extension Clear	E=0
EQ	EQual	Z=1
ES	Extension Set	E=1
GE	Greater or Equal	(N eor V) = 0
GT	Greater Than	Z or (N eor V)=0
HS	Higher or Same	C=0
LC	Limit Clear	L=0
LE	Less or Equal	Z or (N eor V)=1
LS	Limit Set	L=0
LT	Less Than	(N eor V) = 1
MI	Minus	N=1
NE	Not Equal	Z=0
NR	NoRmalized	Z+(not U and not E)=1
PL	Plus	N=0
NN	Not Normalized	Z+(not U and not E)=0

7.4 Ansteuern des DSP's über das Betriebssystem

Hier folgen die Systemaufrufe, die ATARI zur Programmierung des DSP ins Falcon-TOS implementiert hat.

Nomenklatur:

BYTE = 8 Bit
WORD = 16 Bit
LONG = 32 Bit
VOID = Dummy, d.h. keine Rückgabe
* = Zeiger auf

Rückgabe erfolgt in D0

Beispiel:

XBIOS 110

VOID DSP_ExecBoot(LONG *codeptr, LONG codesize,
WORD ability)

ist

```
move.w   #ability,-(sp)
move.l   #codesize,-(sp)
pea      codeptr
move.w   #110,-(sp)
trap     #14
lea      12(sp),sp
```

7.4.1 Daten Transfer Routinen

XBIOS 96

```
VOID Dsp_DoBlock(    LONG *data_in, LONG size_in,
                     LONG *data_out, LONG size_out )
```

Diese Funktion dient zum Übertragen von Daten zwischen den Prozessen auf der CPU und dem DSP.

*data_in , Feld mit den zu übertragenden Daten als Inhalt
size_in , Menge 64K der zu übertragenden DSP Wörter
*data_out, freies Feld für Daten vom DSP
size_aut , Menge 64K der vom DSP zu übertragenden DSP Wörter

XBIOS 97

```
VOID Dsp_BlkHandShake(    LONG *data_in, LONG size_in,
                          LONG *data_out, LONG size_out )
```

Diese Funktion dient zum Übertragen von Daten zwischen den Prozessen auf der CPU und dem DSP. Ist also identisch zur Dsp_DoBlock() bis auf die Art der Übertragung, die in diesem Falle im Handshaking erfolgt.

*data_in , Feld mit den zu übertragenden Daten als Inhalt
size_in , Menge 64K der zu übertragenden DSP Wörter
*data_out , freies Feld für Daten vom DSP
size_aut , Menge 64K der vom DSP zu übertragenden DSP Wörter

XBIOS 98

VOID Dsp_BlkUnpacked(LONG *data_in, LONG size_in,
LONG *data_out, LONG size_out)

Noch eine Routine zum Übertragen von Daten zwischen DSP und CPU Prozessen. Bei diesem Befehl werden die DSP Wörter als Langwörter übertragen, d.h. Dsp_GetWordSize muß 4 sein. Wenn die übertragenen DSP Wörter < 4 sind, ist der Inhalt der nicht übertragenen Teile des Langworts nicht definiert.

*data_in , Feld mit den zu übertragenden Langwörtern
size_in , Menge 64K der zu übertragednen DSP Wörter
*data_out, freies Feld für Langwörter vom DSP
size_aut , Menge 64K der vom DSP zu übertragenden Langwörter

XBIOS 99

VOID Dsp_InStream(LONG *data_in, LONG block_size,
LONG num_blocks, LONG *block_done)

Überträgt Daten zum DSP via Interrupt. Ein Interrupt zeigt an, daß der DSP bereit ist, Daten zu empfangen. Die Übertragung erfolgt nicht per Handshake, sondern die Funktion liefert bei jedem DSP-Interrupt Daten zum DSP.

data_in , Zeiger auf zu übertragende Blöcke
block_size , Menge der pro Block zu übertragenden DSP Wörter
num_blocks , Menge der zu übertragenden Blocks
block_done , Zeiger auf eine Variable, in die bei Beendigung hineingeschrieben wird, wieviel Blöcke übertragen wurden

XBIOS 100

VOID Dsp_OutStream(LONG *data_out, LONG block_size,
LONG num_blocks, LONG *block_done)

Ähnlich Dsp_InStream, mit dem Unterschied, daß die Daten von DSP kommen und nicht zu ihm übertragen werden.

data_out ,Zeiger auf ein Feld für ankommende Daten
block_size , Menge der zu empfangenden DSP Wörter
num_blocks, Menge der zu empfangenden Blocks
block_done , Zeiger auf eine Variable, in die bei Beendigung hineingeschrieben wird, wieviel Blöcke übertragen wurden

XBIOS 101

VOID Dsp_IOStream(LONG *data_in, LONG *data_out,
LONG block_insize, LONG block_outsize
LONG num_blocks, LONG *block_done)

Ähnlich Dsp_Instream, allerdings wird bei jedem zum DSP übertragenen BLock ein Block vom DSP abgeholt.

data_in , Zeiger auf zu übertragende Blöcke
data_out , Zeiger auf ein Feld für ankommende Daten
block_outsize ,Menge der zu empfangenden DSP Wörter
block_insize , Menge der pro Block zu übertragenden DSP Wörter
num_blocks , Menge der zu empfangenden Blocks
block_done , Zeiger auf eine Variable, in die bei Beendigung hineingeschrieben wird, wieviel Blöcke übertragen wurden

XBIOS 102

VOID Dsp_RemoveInterrupt(WORD mask)

Funktion zum Ausschalten der DSP-Interrupts, sollte aufgerufen werden, wenn eine der "stream" Aufrufe weniger Daten als erwartet übertragen hat.

mask, %XXXXXXtr

r = kein Interrupt, wenn der DSP bereit ist zu empfangen
t = kein Interrupt, wenn der DSP bereit ist zu senden

XBIOS 103

WORD Dsp_GetWordSize()

Gibt die Größe eines DSP-Worts in Bytes zurück (momentan 3 Bytes). Dieser Aufruf sollte auf jedem Fall vor der Benutzung von Datentransfer-Funktionen gemacht werden, um kompatibel zu kommenden ATARI Computern zu bleiben.

XBIOS 123

VOID Dsp_BlkWords(LONG *data_in, LONG size_in,
LONG *data_out, LONG size_out)

Again, eine Routine zum Übertragen zwischen DSP und CPU Prozessen. Allerdings hier nur vorzeichenbehaftete 16 Bit Werte, die vorzeichenrichtig auf die Wortbreite des DSP erweitert werden. Vom DSP kommende Worte werden ebenso wieder auf signed 16 Bit Werte gebracht, d.h. nur das mid&low Byte werden vom DSP übertragen.

*data_in , Feld, mit den zu übertragenen Vorzeichen behafteten

16 Bit Worten
size_in , Menge 64K der zu übertragenden Wörtern
*data_out , freies Feld mit Platz für die Wörter
size_aut , Menge 64K der vom DSP zu übertragenden Wörter

XBIOS 124

VOID Dsp_BlkBytes(LONG *data_in, LONG size_in,
LONG *data_out, LONG size_out)

Es reißt nicht ab, wiederum eine Routine zum Datenaustausch zwischen DSP und CPU Prozessen. Diesmal werden die Daten byteweise übertragen.

*data_in , Feld mit den zu übertragenden Bytes
size_in , Menge 64K der zu übertragenen Bytes
*data_out , freies Feld für Bytes vom DSP
size_aut , Menge 64K der vom DSP zu übertragenden Bytes

XBIOS 126

VOID Dsp_SetVectors(LONG *receiver, LONG *transmitter)

Installieren eigener Interruptroutinen, zum Senden oder Empfangen vom/zum DSP. Wenn 'transmitter' <> 0 die niederwertigen 24 Bit von 'transmitter' werden zum DSP gesendet, ansonsten bekommt der DSP keine Meldung. Wenn beide Zeiger Null werden, werden keine Interruptroutinen installiert.

receiver , Zeiger auf Routine zum Verarbeiten der vom DSP kommenden Daten
0 = nicht installieren
transmitter , Zeiger auf Routine zum Senden von Daten zum DSP
0 = nicht installieren

XBIOS 127

Diese Funktion kann im Prinzip das, was alle anderen Funktionen zuvor konnten. Allerdings kann man mit dieser Funktion mehrere Typen von Daten von/in unterschiedliche Speicherbereiche von/in den DSP übertragen. Um das zu koordinieren, muß eine Struktur angelegt werden, die Art, Menge und Ort wohin bzw. wovon die Daten kommen. Man spart sich mit diesem Aufruf unter Umständen viele Aufrufe der zuvor genannten Funktionen.

```
VOID Dsp_MultBlocks(  LONG numsend, LONG numreceive,
                      LONG *sendblocks, LONG *receiveblocks)
```

numsend , Menge der zu sendenden DSP Blockelemente
numreceive , Menge der zu empfangenden DSP BLockelemente
*sendblocks , Zeiger auf eine Struktur der zu sendenden Daten
*receiveblocks , Zeiger auf eine Struktur der zu empfangenden Daten

Struktur:

WORD blocktype , Art der Daten im Block
0 = Langworte, 1 = vorzeichenbehaftete Worte
2 = Bytes

LONG blocksize , Menge der Elemente im Block
LONG *blockaddr , Adresse des Blocks

7.4.2 Programm Kontroll Routinen

XBIOS 104

LONG Dsp_Lock()

Dieser Aufruf ist so 'ne Art Semaphor, sie ermöglicht es einem Programm herauszufinden, ob der DSP gerade von einer anderen Applikation benutzt wird.

Rückgabe: 0 = DSP frei
- 1 = DSP benutzt

XBIOS 105

VOID Dsp_Unlock()

Um den DSP für andere Applikationen wieder frei zu geben, nach dem man ihn zuvor mit 'Dsp_Lock' gesperrt hat, muß diese Funktion aufgerufen werden

XBIOS 106

VOID Dsp_Avaiable(LONG *xavaiable, LONG *yavaiable)

Gibt Menge des vorhandenen X- und Y-Speichers aus. Da der P Speicher den X- und Y-Speicher überlagert, liegen in den unteren 64*24 Bit des Y-Speichers die DSP Interrupt Vektoren.

xavaiable , Zeiger auf eine Variable, in die die Menge des verfügbaren X Speichers eingetragen wird

yavaiable , Zeiger auf eine Variable, in die die Menge des verfügbaren Y Speichers eingetragen wird

XBIOS 107

LONG Dsp_Reserve(LONG xreserve, LONG yreserve)

Der angeforderte Speicher sollte die von 'Dsp_Avaiable' erhaltenen Werte nicht übersteigen. Der Speicherbereich wird solange vor Überschreiben geschützt, bis der 'Dsp_Reserve' Aufruf nochmals erfolgt.

xreserve, Menge des zu reservierenden X-Speichers in DSP Worten
yreserve, Menge des zu reservierenden Y-Speichers in DSP Worten

gibt bei erfolgreichem Aufruf 0 zurück

XBIOS 108

LONG Dsp_LoadProg(LONG *file, WORD ability, LONG *buffer)

Lädt Programme im ".LOD" Format, wandelt sie vom ASCII ins binär Format und führt sie in den durch 'Dsp_Reserve" reservierten Bereich im DSP aus.

file , Zeiger auf den Dateinamen
ability , Kennzeichnung des Programms
buffer , Zeiger auf Zwischenspeicher für den generierten DSP Code

die Größe des Speicherbereichs, die auf den Puffer zeigt, berechnet sich wie folgt:

wort = Dsp Wortgröße
anz = Anzahl der Programm und Datenworte in der ".LOD" Datei
block = Anzahl der Blöcke in der ".LOD" Datei

wort*(anz+(wort*block))

Gibt eine Null zurück, falls der Aufruf erfolgreich verlaufen ist, ansonsten -1.

XBIOS 109

VOID Dsp_ExecProg(LONG *codeptr, LONG codesize,
WORD ability)

Ähnlich wie 'Dsp_LoadProg', mit dem Unterschied, daß das Programm nicht geladen und ins Binär-Format umgewandelt werden muß, sondern in dieser Form schon im Speicher erwartet wird.

codeptr , Zeiger auf ein DSP Programm im Binär-Format
codesize , Länge des Programms
ability , Kennzeichnung des Programms

XBIOS 110

VOID Dsp_ExecBoot(LONG *codeptr, LONG codesize, WORD ability)

Diese Funktion führt einen Reset im DSP durch, lädt ein Programm in das interne 512 Wort Breite RAM des DSP und startet es. Laut ATARI sollte dieser Aufruf nur von Debuggern oder ähnl. Programmen verwendet werden, andere Programme sollten nicht den DSP komplett für sich in Anspruch nehmen.

codeptr , Zeiger auf ein DSP Programm im Binär-Format
codesize , Länge des Programms < 512 DSP Worte
ability , Kennzeichnung des Programms

XBIOS 111

Dieser Aufruf lädt ".LOD" Dateien und wandelt sie ins Binär-Format um.

LONG Dsp_LodToBinary(LONG *file, LONG *codeptr)

file , Zeiger auf den Dateinamen
codeptr , Zeiger auf den Code im ".LOD" ASCII-Format

Der Speicherbereich auf den 'codeptr' zeigt, muß groß genug gewählt werden, um das DSP Programm im Binär-Format aufnehmen zu können.

Die Funktion liefert die Größe des Programms im Binär-Format zurück, im Fehlerfall eine negative Zahl.

XBIOS 112

VOID Dsp_TriggerHC(WORD vector)

Starten einer Routine via Host Command Vektor des DSP. Nur die Vektoren $13 und $14 sind frei, alle anderen sind durch das Betriebssystem belegt und werden bei jedem Programmstart im DSP wieder initialisiert (siehe 7.2.2).

vector, Host Command Vektor des DSP ($13 u. $14)

XBIOS 113

WORD Dsp_RequestUniqueAbility()

Gibt eine einmalige 16 Bit Nummer als Kennzeichnung (ability) im DSP zurück, diese kann man dann zum Laden eines Programms in den DSP benuzen. Bei einem erneuten Versuch, das DSP Pro-

gramm zu starten, braucht man dann nur zu testen, ob es noch vorhanden ist. Das macht es unnötig, vor jedem Start einer Routine oder Programms im DSP diese nachzuladen.

XBIOS 114

WORD Dsp_GetProgAbility()

Gibt zurück, welche Kennzeichnung (ability) das Programm, das sich im Moment im DSP befindet, besitzt.

XBIOS 115

VOID Dsp_Flushroutines()

Löscht Unteroutinen aus dem DSP Speicher, sollte nur wenn unbedingt nötig benutzt werden, um, wenn mehrere Prozesse den DSP benutzen, das zeitraubende dauernde Neueinladen von Unterroutinen zu vermeiden.

XBIOS 116

WORD Dsp_LoadSubroutine(LONG *ptr, LONG size, WORD ability)

Lädt eine Unterroutine in den DSP Speicher, die darin resident verbleibt. Erst wenn alle Plätze für Unterroutinen belegt sind, kann es mit einer anderen Routine überschrieben werden. Auch kann man eine DSP-Unterroutine mit Dsp_FlushSubroutine aus dem Speicher entfernen.

ptr , Zeiger auf den Code im Binär-Format
size , Länge der Routine in DSP Worten
ability , Kennzeichnung der Routine

Gibt ein positives Handle zurück, wenn die Routine erfolgreich im Speicher untergebracht, Null wenn sie nicht installiert wurde.

XBIOS 117

WORD Dsp_InqSubrAbility(WORD ability)

Gibt das Handle der jeweiligen Unterroutine zurück. Damit ein Prozess Unterroutinen benutzen kann, die er selbst gar nicht installiert hat, z.B. Unterroutinen, deren Kennzeichnung ATARI festgelegt hat, also eine Art Library-Funktion, die der Prozess sonst selbst geladen hätte.

ability, Kennzeichnung der Routine

XBIOS 118

WORD Dsp_RunSubroutine(WORD handle)

Startet im DSP Speicher befindliche Unterroutine.

handle, Wert den man z.B. von 'Dsp_LoadSubroutine' zurück bekommt

Eine Null gibt an, daß die Unterroutine erfolgreich gestartet wurde, ein negativer Wert, daß ein Fehler aufgetreten ist.

XBIOS 119

WORD Dsp_Hf0(WORD flag)

Ändern von Bit 3 im Host Status Register (HSR) des DSP.

flag, 0 = HF0 löschen, 1 = HF0 setzen, -1 = nicht ändern

Gibt alten Inhalt zurück.

XBIOS 120

WORD Dsp_Hf1(WORD flag)

Ändern von Bit 4 im Host Status Register (HSR) des DSP.

flag, 0 = HF1 löschen, 1 = HF1 setzen, -1 = nicht ändern

Gibt alten Inhalt zurück.

XBIOS 121

WORD Dsp_Hf2(WORD flag)

Ändern von Bit 3 im Host Control Register (HCR) des DSP.

flag, 0 = HF2 löschen, 1 = HF2 setzen, -1 = nicht ändern

Gibt alten Inhalt zurück.

XBIOS 122

WORD Dsp_Hf3(WORD flag)

Ändern von Bit 4 im Host Control Register (HCR) des DSP.

flag, 0 = HF3 löschen, 1 = HF3 setzen, -1 = nicht ändern

Gibt alten Inhalt zurück.

XBIOS 125

BYTE Dsp_HStat()

Gibt den Inhalt des Interrupt Status Registers (ISR) des DSP zurück. Gibt an, ob der Host Port zum Senden oder Empfangen von Daten bereit ist.

7.5 Ideen zum DSP

Wie oben schon erwähnt, hatte ich den Falcon nur sehr kurze Zeit, ich konnte mich zwar recht ausgiebig mit ihm beschäftigen, aber der DSP ist einfach zu komplex, um alles mit ihm Mögliche in kurzer Zeit durchspielen zu können. Also hier noch einige Ideen zum Nutzen des Falcon für Spieleprogrammierer.
Da der DSP im Falcon030 schneller als die CPU ist, könnte ich mir gut vorstellen, daß einige Programmierer auf die Idee kommen, rechenintensive Spiele komplett im DSP laufen zu lassen und nur die audiovisuelle Präsentation auf dem Rest des System laufen zu lassen.
Auch müßte es meiner Meinung nach möglich sein, aus dem Falcon mit Hilfe des DSP, achtstimmige Samples mit variabler Abspielfrequenz auf dem CODEC D/A Wandler auszugeben. Dazu muß man per DMA dem DSP die Sounds häppchenweise zum Zusammenmischen übergeben, der speichert diese zwischen und mischt sie im angemessenen Frequenzverhältnis.
Ebenfalls könnte man per DMA gepackte Bilddaten in den DSP laden und Stück für Stück entpackt und per DMA in einen anderen Speicherbereich übertragen, während der User auf dem Bildschirm "fröhlich" weiterarbeiten kann. Aber nicht nur entpacken kann der DSP, auch das Manipulieren und Interpolieren von Bilddaten könnte man den DSP erledigen lassen.
Auch einen mathematischen Coprozessor könnte man den DSP emulieren lassen und via LINE F die Coprozessor Befehle emulieren. Das wäre dann zwar, durch die Zeit, die bei den LINE F Exceptions verloren gehen, nicht besonders schnell, aber immer noch schneller als wenn der 68030 mit 16MHz die Emulation vornehmen sollte.
WIchtig! ist, der DSP ist ein eigenes System mit eigenem Bus, d.h. alles was an Code im DSP läuft, läuft volkommen unabhängig von Busmastern auf dem 68030 Daten/Adressbus. Selbst wenn Floppy oder SCSI DMA läuft, kann der DSP (falls er kein Datentransfer auf/vom System Bus stattfindet) ungestört weiterarbeiten.

Anhang A Optimierung

Bei Optimierungen muß man höllisch aufpassen, da die Programme danach meist viel schlechter lesbar sind. Man sollte daher nur sehr zeitkritische Programmteile auf Geschwindigkeit optimieren und nur wirklich speicherplatzkritische auf Länge.

A.1 Hochsprachen Optimierung

A.1.1 Wo ein 'IF' ist, darf auch ein 'ELSE' sein

Ein häufiger Fehler von Programmieranfängern ist, daß sie bei mehreren ausschließenden 'IF' Abfragen diese untereinander setzen und nicht nach der ersten Abrage die restlichen in 'ELSE' Zweige.

Beipiel: FALSCH !!!

Geht alle Bedingungen durch, obwohl möglicherweise schon die erste die wahre Aussage ist !

```
IF x%=4 AND y%=1
  PRINT "TOTAL"
ENDIF
IF x%=3 AND y%=4
  PRINT "NOJO"
ENDIF
IF x%=8 AND y%=3
  PRINT "OJOJ"
ENDIF
```

Beispiel: RICHTIG !!!

```
IF x%=4 AND y%=1
  PRINT "TOTAL"
ELSE IF x%=3 AND y%=4
  PRINT "NOJO"
ELSE IF x%=8 AND y%=3
  PRINT "OJOJ"
ENDIF
```

Im zweiten Fall werden nach einer richtigen Bedingung die restlichen Vergleiche gar nicht mehr abgefragt. Wenn man die Vergleiche dann noch nach möglicher Häufigkeit ordnet, d.h. als allererstes die Bedingung abfragen, die öfter als die anderen kommen könnte und den Rest in wahrscheinlicher nach unwahrscheinlicher ordnet, kann man an zeitkritischen Stellen z.B. Kollision einiges an Rechenzeit sparen.

A.1.2 Es müssen nicht immer Fließkommazahlen sein !

Bei Spielen z.B. Simulationen werden oft irgendwelche Formeln mit vielen "komplizierten" Ausdrücken berechnet, die dann aber meist nur pixelgenau grafisch dargestellt werden. In solchen Fällen hat es sich in Assembler als vorteilhaft erwiesen, mit geringerer Genauigkeit zu rechnen z.B Langworten und das niederwertige Wort als Nachkommastellen und das höherwertige als ganze Zahlen zu nutzen. So kann man sich dann die komplizierte Fließkommaroutinen sparen.
In Hochsprachen kann man sich mit einer ähnlichen Technik in vielen Fällen die Fließkommazahlen sparen. Man braucht z.B. Zahlen mit einer Nachkommastelle, dann nimmt man die Werte mit denen man rechnen möchte, multipliziert sie mal zehn und überträgt diese z.B. in 32 Bit Variablen. Der Wert vor dem Komma kann dann immer noch bis über 200 Millionen groß werden, was für die meisten Fälle genügen dürfte. Allerdings gibt es bestimmte Regeln die man bei der Benutzung dieser Zahlen beachten muß.

Addieren:

kein Proplem

Multiplizieren:

Wenn 2 gewandelte Zahlen multipliziert werden, muß das Ergebnis durch den Faktor geteilt werden, mit dem sie umgewandelt wurden.

Beispiel:

```
n=100.23

n%=n*100 ! also 10023

k=450.33

k%=k*100 ! also 45033
```

'nimmt man k%*2, kann man das Ergebnis am Ende wieder durch 100 teilen und bekommt dann k*2

```
k%=k%*2 !k%=90066

k=k%/100 !k=900,66 also genau 2*k
```

'allerdings wenn man k% * n% nimmt

```
k%=k% * n% ! 90066*10023=902731518
' Wenn man das durch 100 teilt, hat man eben dann noch nicht
' das Ergebnis

k=k%/10000 ! das ergibt 90273.1518, also genau k*n
```

A.1.3 Addieren ist oftmals besser als Multipizieren !

Beim Ausrechnen einer Formel per Hand ist es im Prinzip egal, ob der Taschenrechner eine Zehntelsekunde schneller oder langsammer "mit dem Ergebnis herausrückt", da in diesem Fall der Mensch das langsamste Glied in der Kette ist und somit jeder zusätzliche Denkprozeß die Zeit, die zum Errechnen eines Ergebnisses benötigt, erhöhen würde. Bei einem Programm ist das umgekehrt, denn wenn der Algorithmus fertig ist, läßt jede zusätzliche Optimierung (also Denkprozeß des Programmierers) das Programm etwas schneller das Ziel (also die Lösung) erreichen.

Hier ein Beispiel für Optimierung:

```
C=2*B
D=3*B
```

eine Optimierung wäre in diesem Fall,

```
C=2*B
D=C+B
```

da eine Addition von Rechnern in der Regel schneller durchgeführt wird als eine Multiplikation.
Nicht immer läßt sich eine Optimierung so leicht erkennen wie in diesem Fall und es ist auch nicht immer sinnvoll, bis zur letzten Konsequenz einen Algorithmus zu optimieren, wenn ein Programm eh schon schnell genug ist und die Lesbarkeit des Programms nicht mehr gewährleistet ist (das ist bei Spielen aber nicht so wichtig, da Spielprogramme nur in seltenen Fällen von anderen Programmierern gewartet (d.h. Fehler im veröffentlichten Programm ausbessern) werden.

A.1.4 Erst rechnen, dann vergleichen

Vergleiche kommen sehr häufig in Programmen vor und oft muß man ein und denselben Ausdruck mit vielen anderen Ausdrücken vergleichen. Wenn der Ausdruck jedoch eine Formel ist, sollte man sich überlegen, ob man nicht erst rechen und dann vergleichen sollte.

Beispiel:

```
IF 2*(x+z)<2*n+4 or  2*(x+z)<100
  PRINT "ZIP ZIP BRRRR"
ENDIF
```

schneller !

```
k=2*(x+z)
IF k<2*n+4 or k<100
  PRINT "ZIP ZIP BRRRR"
ENDIF
```

A.2 Assembler Optimierung

Assembler ist die Programmiersprache, in der man die schnellsten Programme schreiben kann, aber die Betonung liegt in diesem Fall auf kann. Man kann in Assembler auch Programme schreiben, die langsamer sind als die compilierte Hochsprache.

Beispiel 1

Das Füllen des Stacks mit Parametern:

```
move.w   #1,-(SP)
move.w   #34,-(SP)
```

kann optimiert werden durch

```
move.l #$340001,-(SP) ; 1 Wort weniger
```

oder

```
pea  $340001   ; 1 Wort weniger
```

falls das zu übertragene Langwort nur noch Wortgröße hat

```
move.w   #1234,-(SP)
clr.w    -(SP)     ; ist scheller wie move.w #0,-(SP)
```

kann optimiert werden durch

```
pea  1234.w   ; 2 Worte weniger
```

Beispiel 2

Bereinigen des Stacks:

```
add.l #4,SP
```

schneller

```
addq.l #4,SP ;soweit ist es noch offensichtlich
```

aber

```
add.l #12,SP
```

4 Taktzyklen schneller

```
lea 12(SP),SP
```

Beispiel 3

Um ein Bit nach links schieben:

```
lsl.l #1,D0
```

schneller

```
add.l D0,D0
```

Löschen von Registern:

```
clr.l D0
```

kann optimiert werden durch

```
sub.l D0,D0    ; schneller !
```

Generell gilt, je weniger Zugriffe auf den Speicher desto schneller. Das heißt nichts anderes, als daß man so viel Daten wie möglich in den Registern halten sollte (davon profitieren ja auch die RISC Prozessoren). Oft benutzte Daten in Registern halten, da jeder Speicherzugriff längeren Code erzeugt und, wie oben schon erwähnt, länger dauert.
Wenn es schnell gehen soll, vermeidet man besser Schleifen und führt die Befehle mit den einzelnen Parametern hintereinander aus. Es fallen so die Sprünge und Vergleiche weg. Übrigens das funktioniert auch in Hochsprachen (unrolling LOOPS).

Anhang B

Quellen und weiterführende Literatur

Bücher:

Jankowski,Rabich,Reschke:Atari Profibuch ST-STE-TT
SYBEX, Düsseldorf 1987/88/91/92 ISBN 3-88745-888-5

Das Standardwerk für Programmierer auf dem ATARI.

Klein,Thiel:i860 Mikroprozessor der Superklasse
Francis, München 1991 ISBN 3-7723-4191-8

Man erfährt darin auch einiges über den 68040 und Prozessoren schlechthin.

Williams:68030 assembly language reference
Addison Wesley, USA 1989 ISBN 0-201-08876-2

Komplette Einführung in die 68000 bis 68030 und 68881/2 Prozessoren, auf jeden Fall empfehlenswert.

Computer graphics: principles and practice
von James F Foley, van Dam, Feiner, Huges 2nd Edition
Addison Wesley, ISBN 0-201-12110-7

Enthält Informationen über Vektorgrafik, Raytracing u.v.m. Auch ein Standardwerk!

Mader:Einführung in Forth 83
Heim, Darmstadt 1989 ISBN 3-923250-69-X

Gut für Anfänger, auch wenn sie nicht unbedingt Forth programmieren wollen.

DSP 56000/56001 User's Manual
MOTOROLA 1990

Für Leute, die mehr über den DSP wissen wollen.

MC 68040, 32Bit Microprocessors User's Manual
MOTOROLA 1989

Der Falcon040 kommt (hoffentlich) bestimmt !

Peter Wollschlaeger:Atari-ST-Assembler Buch
Markt & Technik, Haar bei München 1987 ISBN 3-89090-467-X

Gut verständliche Einführung in 68000 Assembler, für Anfänger empfehlenswert.

Martin Pohl, Holger Eriksdotar: Die wunderbare Welt der Grafikformate
Wolfram's Fachverlag, ISBN 3-925328-05-X

Zeitschriften:

ST COMPUTER - Heim Verlag
Div. Ausgaben von TOS - ICP Verlag
ST Magazin - Markt&Technik bzw. AWi Verlag
CHIP Special, Animation auf dem PC

Dokumentation von ATARI:

Falcon030 Developer Documentation vom 1.Oktober 1992
THE TT030 COMPANION

anderes:

Anleitung zu NoiseTracker v. 1.1

Anhang C

Inhalt der Diskette !

Der Inhalt der Diskette kann sich je nach Auflage ändern, auf ihr befinden sich aber Dateien, die sich nicht ändern !

INHALT.TXT , Inhaltsangabe der Diskette
FEHLER.TXT, bekannte Fehler des Buches

und einige Files mit dem Kennzeichner '.TOS' und den Namen 'KAP' gefolgt von 'A' oder 'B' und einer Kapitelnummer. Diese Files sind selfextracting, d.h. sie entpacken sich selbst. Man muß diese Files nur je auf eine extra Diskette kopieren und starten, dann werden aus den jeweiligen '.TOS' Programmen die Dateien (Listings, Programme etc.) zum jeweiligen Kapitel.

Nachwort:

"Zu Risiken und Nebenwirkungen fragen Sie Ihren Arzt oder Apotheker."

Nun, über ein Jahr habe ich für dieses Buch gebraucht (ich mußte ja zwischendurch noch studieren). Ich hoffe, mein Ziel, zu zeigen was in den ATARI Computern steckt und wie man dessen Resourcen richtig (!) nutzt, habe ich erreicht. Aber das wird wohl erst die Resonanz auf mein Buch zeigen, das mein Debut, was Bücherschreiben betrifft, ist.

Der Falcon030 hat mich sozusagen kalt erwischt, ich habe trotzdem alle möglichen Informationen gesammelt und in dieses Buch mit aufgenommen, so daß dieses Buch für den, der auf dem Falcon Spiele programmiert, eine Fundgrube sein wird (nehme ich doch stark an !).

Auf dem TT ist mehr machbar als man denkt. Bitte, liebe Spieleprogrammierer, schenkt diesem Gerät die Beachtung, die ihm gebührt.

"Juhu, mein erstes Buch ist fertig ! (endlich)"

Kritik und Anregungen bitte an den Verlag, der diese an mich weiterreichen wird !

"Geldwünsche und/oder Drohungen sinnlos, habe selbst nichts ! "

Beantwortet werden nur Briefe von weiblichen "Fans" oder Post mit beigefügtem, frankiertem und an sich selbst adressierten Rückumschlag.

oder über E - MAIL

s=Pollack; ou=rz; p=fh-frankfurt; a=dbp; c=de

P.S. Jetzt programmiert bitte viele tolle Spiele für Euren ATARI !!!

Stichwortverzeichnis

Spiele selbst programmieren

Wichtige Merkmale:

Wer bisher dem interessierten Anfänger ein Buch zur Spieleprogrammierung für den ATARI Rechner nennen sollte war überfragt. Dieses Buch beseitigt dieses Manko. Es zeigt dem interessierten Anfänger, was er bei der Programmierung seines ersten Spieles beachten sollte und erklärt Schritt für Schritt den Aufbau eines Spieles anhand vieler Beispiele. Dem Fortgeschrittenen stellt das Buch das entsprechende Know-How zur Verfügung, um professionelle Effekte, wie man sie von guten Spielen kennt, in seinen eigenen Programmen zu verwenden. Für alle Umsteiger von anderen Rechnersystemen hilft es gewohnte Effekte auf den Atari zu konvertieren.

Aus dem Inhalt:

- ❒ Einführung in BASIC und Assembler
- ❒ Mehrfarbige Sprites
- ❒ Ein komplettes Spiel - Aufbau und Programmierung
- ❒ Kompatible Programmierung von ST/STE/TT/Falcon 030
- ❒ Programmierung der Grafikhardware
- ❒ Rasterinterrupts
- ❒ Soundprogrammierung, PSG, STE/TT/Falcon 030 DMA
- ❒ Format der AMIGA Soundmodule
- ❒ Diverse Grafikformate
- ❒ Nutzung des DSP 56001 bei der Spieleprogammierung
- ❒ MIDI - Programmierung
- ❒ und vieles mehr

ISBN 3-928480-13-8
Bestell-Nr. B-465
DM 59,-

incl. Diskette